

Spring Batch 批处理框架

刘相 编著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书全面、系统地介绍了批处理框架 Spring Batch，通过详尽的实战示例向读者展示了 Spring Batch 框架对大数据批处理的基本开发能力，并对框架的架构设计、源码做了特定的剖析；在帮助读者掌握 Spring Batch 框架基本功能、高级功能的同时，深入剖析了 Spring Batch 框架的设计原理，帮助读者可以游刃有余地掌握 Spring Batch 框架。

本书分为入门篇、基本篇和高级篇三部分。入门篇介绍了批处理、Spring Batch 的基本特性和新特性，快速入门的 Hello World 等内容引领读者入门，从而进入数据批处理的世界。基本篇重点讲述了数据批处理的核心概念、典型的作业配置、作业步配置，以及 Spring Batch 框架中经典的三步走策略：数据读、数据处理和数据写，详尽地介绍了如何对 CVS 格式文件、JSON 格式文件、XML 文件、数据库和 JMS 消息队列中的数据进行读操作、处理和写操作，对于数据库的操作详细介绍了使用 JDBC、Hibernate、存储过程、JPA、Ibatis 等处理。高级篇提供了高性能、高可靠性、并行处理的能力，分别向读者展示了如何实现作业流的控制，包括顺序流、条件流、并行流，如何实现健壮的作业，包括跳过、重试和重启等，如何实现扩展作业及并行作业，包括多线程作业、并行作业、远程作业和分区作业等，从而实现分布式、高性能、高扩展性的数据批处理作业。

本书适合需要具体使用批处理作业、大数据处理的开发人员，设计人员和架构师，对于企业中存在大量作业的运维人员亦有一定的参考价值。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Spring Batch 批处理框架 / 刘相编著. —北京：电子工业出版社，2015.2
ISBN 978-7-121-25241-9

I. ①S… II. ①刘… III. ①数据处理 IV. ①TP274

中国版本图书馆 CIP 数据核字（2014）第 302744 号

策划编辑：孙学瑛

责任编辑：徐津平

特约编辑：顾慧芳

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：25.25 字数：582 千字

版 次：2015 年 2 月第 1 版

印 次：2015 年 2 月第 1 次印刷

印 数：3000 册 定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

批处理编程之美

这是一部论述批处理程序编程的书。在信息系统中，联机和批处理是计算机处理的两种基本模式，前者快速响应、超时中断、密集并发，后者处理时间长、异常需要支持重做、通常以顺序执行。早期的计算机系统多采用批处理的处理模式，客户机/服务器架构的产生让联机模式越来越多地被采用，但批处理模式一直在信息系统中起着重要作用，随着 OLAP、大数据等新技术的应用，批处理的处理模式又成为热点，例如在传统银行 IT 系统中每日动辄运行上万个批处理作业，在互联网应用中，腾讯、阿里每日的批处理作业可达百万量级的水平。

编程之美，美在架构，架构之美，美在抽象，只有具备充分理解复杂业务场景的格局，才能进行将复杂问题做简单化的抽象。同联机模式汗牛充栋的著作、框架相比，批处理模式的抽象不多，著名的 MapReduce 就是其中之一，MapReduce 将大批数据的处理过程进行了抽象，而 Spring Batch 则是对编写批处理程序本身的特性进行了抽象。通过将批处理程序分解为 Job 和 Job Step 两个部分，将处理环节定义为数据读、数据处理和数据写三个步骤，将异常处理机制归结为跳过、重试、重启三种类型，将作业方式区分为多线程、并行、远程、分区四大特征，正所谓增一分则肥，减一分则瘦。类似之美，Spring 系列项目还有很多，例如 Spring Framework 对依赖注入的抽象，Spring integration 中利用消息、队列、处理器三个概念的组合对集成模式的抽象，都让我叹为观止。

当相相^①递给我他的新作时，我吃了一惊，惊在他“悄无声息”地完成了这样一个大部头作品，迫不及待地有一种先睹为快的冲动。我发现，书中通过对 Spring Batch 本身的论述，让我体会到了 Spring Batch 的精髓，也更加深刻地理解了批处理编程模式，还看到了相相对信息系统中如何使用这一框架的见解，毕竟信息系统中的批处理程序，不仅仅是一个框架，还需要包含更多的管理、运维方面的流程、制度与平台。近年来，我看到很多企业都在构建集中的批处理平台，管理大量出现的批处理作业，也期待相相能有更多这方面的分享。

普元 CTO 焦烈焱

2015 年 1 月

① 相相为本书作者刘相的简称。

前言

信息时代，数据是现代企业最宝贵的核心资产，是企业运用科学管理、决策分析的基础。截至目前，国内大多数企业已经完成了 OLTP（联机事务处理）的业务系统和办公自动化系统，用来记录事务处理的各种相关数据。据统计，企业的数据每年都在成倍增长，企业如能充分利用这些数据会带来巨大的商机。但目前企业通常所关注的数据仅占总数据量的 5% 左右，企业没有最大化地利用已经存在的数据资源，导致浪费了更多的时间和资金，同时也失去了制定关键商业决策的最佳契机。于是，企业如何通过各种技术手段，并把数据转换为信息、知识和商机已经成为提高其核心竞争力的主要手段。而数据批处理则是达成上述目标的一个主要技术手段，通过数据批量处理，可以完成数据的加载、抽取、转换、清洗等功能，进而支撑企业的各种数据分析。

2012 年底，笔者有幸接手某银行批处理项目，首次接触 Spring Batch 批处理框架，深入学习 Spring Batch 框架后发现：Spring Batch 框架的架构设计清晰和优雅，其功能完备，具有无所不在的扩展能力、丰富的业务组件，并且采用业务与技术分离的设计思想。笔者通过近半年多的学习，通读了官网提供的所有文档，深入学习了原版书籍 *Spring Batch In Action*。在大数据时代，批处理框架在金融、电信、大型制造业等应用非常广泛。在学习的过程中，笔者了解到同事、网络上有不少朋友苦于没有中文版的 Spring Batch 框架介绍资料，于是萌生出写一本 Spring Batch 框架中文版介绍图书的想法，希望能够帮助国内的读者快速地了解该框架。

自 2013 年年中起笔者开始构思本书的大纲，在繁忙工作之余坚持每周完成部分章节内容，前后历时 1 年多时间完成书稿编写工作。在此感谢妻子 Phyllis 给予我充足的时间、感谢可爱女儿 Rachel 给我带来的巨大欢乐动力。本书主要的目的是全面、系统地介绍批处理框架 Spring Batch，通过详尽的实战示例向读者展示了 Spring Batch 框架的基本开发能力，并对框架的架构设计、源码做了特定的剖析；在帮助读者掌握该框架基本功能、高级功能使用的同时，深入剖析 Spring Batch 框架的设计原理，帮助读者可以游刃有余地掌握 Spring Batch 框架。

本书分为入门篇、基本篇、高级篇三个部分，从基本的入门篇讲起，通过介绍批处理、Spring Batch 基本特性、新特性、快速入门的 Hello World 等内容引领读者入门，进入批处理的世界。之后的基本篇，重点讲述了批处理的核心概念、典型的作业配置、作业步配置以及 Spring Batch 框架中经典的三步走策略：数据读、数据处理、数据写，详细介绍了如何对分隔符类型文件、定长类型文件、JSON 格式文件、复杂类型格式文件、XML 文件、数据库、JMS

消息队列中的数据进行读、处理、写操作,对于数据库的操作详细介绍了使用 JDBC、Hibernate、存储过程、JPA, Ibatis 等处理。为了能够让读者更深入地了解 Spring Batch 框架,高级篇提供了高性能、高可靠性、并行处理的能力,分别向读者展示如何实现作业流的控制包括顺序流、条件流、并行流,如何实现健壮的作业包括跳过、重试、重启等,如何实现扩展作业及并行作业包括多线程作业、并行作业、远程作业、分区作业等。

本书适合需要具体使用批处理框架 Spring Batch 的开发人员、设计人员、架构师,对于企业中存在大量作业的运维人员亦有一定的参考价值。

编 者

2014.12.23 于上海浦东

目 录

第 1 篇 入门篇

第 1 章 Spring Batch 简介.....	2	1.5 Spring Batch 2.2 新特性.....	13
1.1 什么是批处理.....	2	1.5.1 Spring Data 集成.....	13
1.2 Spring Batch.....	3	1.5.2 支持 Java 配置.....	13
1.2.1 典型场景.....	3	1.5.3 Spring Retry.....	14
1.2.2 Spring Batch 架构.....	4	1.5.4 Job Parameters.....	14
1.3 Spring Batch 优势.....	4	1.6 开发环境搭建.....	15
1.3.1 丰富的开箱即用组件.....	5	第 2 章 Spring Batch 之 Hello World.....	16
1.3.2 面向 Chunk 的处理.....	5	2.1 场景说明.....	16
1.3.3 事务管理能力.....	5	2.2 项目准备.....	16
1.3.4 元数据管理.....	5	2.2.1 项目结构.....	16
1.3.5 易监控的批处理应用.....	5	2.2.2 准备对账单文件.....	17
1.3.6 丰富的流程定义.....	5	2.2.3 定义领域对象.....	18
1.3.7 健壮的批处理应用.....	6	2.3 定义 job 基础设施.....	18
1.3.8 易扩展的批处理应用.....	6	2.4 定义对账 Job.....	19
1.3.9 复用企业现有 IT 资产.....	6	2.4.1 配置 ItemReader.....	19
1.4 Spring Batch 2.0 新特性.....	6	2.4.2 配置 ItemProcessor.....	21
1.4.1 支持 Java 5.....	7	2.4.3 配置 ItemWriter.....	22
1.4.2 支持非顺序的 Step.....	7	2.5 执行 Job.....	23
1.4.3 面向 Chunk 处理.....	7	2.5.1 Java 调用.....	23
1.4.4 元数据访问.....	11	2.5.2 JUnit 单元测试.....	24
1.4.5 扩展性.....	11	2.6 概念预览.....	26
1.4.6 可配置性.....	12		

第 2 篇 基本篇

第 3 章 Spring Batch 基本概念.....	28	3.2.3 Job Execution.....	34
3.1 命名空间.....	29	3.3 Step.....	35
3.2 Job.....	30	3.3.1 Step Execution.....	37
3.2.1 Job Instance.....	31	3.4 Execution Context.....	38
3.2.2 Job Parameters.....	33	3.5 Job Repository.....	39
		3.5.1 Job Repository Schema.....	39

3.5.2 配置 Memory Job Repository	40	5.3.5 读、处理事务	110
3.5.3 配置 DB Job Repository	41	5.4 拦截器	112
3.5.4 数据库 Schema	42	5.4.1 ChunkListener	115
3.6 Job Launcher	48	5.4.2 ItemReadListener	116
3.7 ItemReader	49	5.4.3 ItemProcessListener	116
3.8 ItemProcessor	50	5.4.4 ItemWriteListener	117
3.9 ItemWriter	50	5.4.5 SkipListener	117
第 4 章 配置作业 Job	52	5.4.6 RetryListener	118
4.1 基本配置	52	第 6 章 读数据 ItemReader	120
4.1.1 重启 Job	54	6.1 ItemReader	120
4.1.2 Job 拦截器	55	6.1.1 ItemReader	120
4.1.3 Job Parameters 校验	58	6.1.2 ItemStream	121
4.1.4 Job 抽象与继承	59	6.1.3 系统读组件	122
4.2 高级特性	61	6.2 Flat 格式文件	122
4.2.1 Step Scope	61	6.2.1 Flat 文件格式	123
4.2.2 属性 Late Binding	62	6.2.2 FlatFileItemReader	125
4.3 运行 Job	63	6.2.3 RecordSeparatorPolicy	129
4.3.1 调度作业	65	6.2.4 LineMapper	130
4.3.2 命令行执行	68	6.2.5 DefaultLineMapper	131
4.3.3 与定时任务集成	71	6.2.6 LineCallbackHandler	138
4.3.4 与 Web 应用集成	73	6.2.7 读分隔符文件	139
4.3.5 停止 Job	77	6.2.8 读定长文件	141
第 5 章 配置作业步 Step	85	6.2.9 读 JSON 文件	143
5.1 配置 Step	86	6.2.10 读记录跨多行文件	145
5.1.1 Step 抽象与继承	87	6.2.11 读混合记录文件	147
5.1.2 Step 执行拦截器	89	6.3 XML 格式文件	150
5.2 配置 Tasklet	92	6.3.1 XML 解析	150
5.2.1 重启 Step	93	6.3.2 Spring OXM	151
5.2.2 事务	94	6.3.3 StaxEventItemReader	153
5.2.3 事务回滚	96	6.4 读多文件	156
5.2.4 多线程 Step	97	6.5 读数据库	159
5.2.5 自定义 Tasklet	97	6.5.1 JdbcCursorItemReader	160
5.3 配置 Chunk	99	6.5.2 HibernateCursorItemReader	167
5.3.1 提交间隔	102	6.5.3 StoredProcedureItemReader	171
5.3.2 异常跳过	103	6.5.4 JdbcPagingItemReader	174
5.3.3 Step 重试	105	6.5.5 HibernatePagingItemReader	179
5.3.4 Chunk 完成策略	107		

6.5.6	JpaPagingItemReader	183	7.8	Item 路由 Writer	254
6.5.7	IbatisPagingItemReader	186	7.9	发送邮件	258
6.6	读 JMS 队列	190	7.9.1	SimpleMailMessageItem Writer	258
6.6.1	JmsItemReader	190	7.10	服务复用	262
6.7	服务复用	194	7.10.1	ItemWriterAdapter	262
6.8	自定义 ItemReader	197	7.10.2	PropertyExtracting DelegatingItemWriter	264
6.8.1	不可重启 ItemReader	197	7.11	自定义 ItemWrite	267
6.8.2	可重启 ItemReader	199	7.11.1	不可重启 ItemWriter	267
6.9	拦截器	202	7.11.2	可重启 ItemWriter	268
6.9.1	拦截器接口	202	7.12	拦截器	271
6.9.2	拦截器异常	203	7.12.1	拦截器接口	271
6.9.3	执行顺序	204	7.12.2	拦截器异常	273
6.9.4	Annotation	204	7.12.3	执行顺序	274
6.9.5	属性 Merge	205	7.12.4	Annotation	274
第 7 章	写数据 ItemWriter	207	7.12.5	属性 Merge	275
7.1	ItemWrite	207	第 8 章	处理数据 ItemProcessor	277
7.1.1	ItemWriter	208	8.1	ItemProcessor	277
7.1.2	ItemStream	208	8.1.1	ItemProcessor	277
7.1.3	系统写组件	209	8.1.2	系统处理组件	278
7.2	Flat 格式文件	210	8.2	数据转换	279
7.2.1	FlatFileItemWriter	210	8.2.1	部分数据转换	279
7.2.2	LineAggregator	214	8.2.2	数据类型转换	281
7.2.3	FieldExtractor	217	8.3	数据过滤	282
7.2.4	回调操作	219	8.3.1	数据 Filter	282
7.3	XML 格式文件	222	8.3.2	数据过滤统计	283
7.3.1	StaxEventItemWriter	222	8.4	数据校验	285
7.3.2	回调操作	226	8.4.1	Validator	285
7.4	写多文件	230	8.4.2	ValidatingItemProcessor	286
7.4.1	MultiResourceItemWriter	230	8.5	组合处理器	288
7.4.2	扩展 MultiResourceItem Writer	233	8.6	服务复用	291
7.5	写数据库	234	8.6.1	ItemProcessorAdapter	291
7.5.1	JdbcBatchItemWriter	235	8.7	拦截器	293
7.5.2	HibernateItemWriter	239	8.7.1	拦截器接口	293
7.5.3	IbatisBatchItemWriter	242	8.7.2	拦截器异常	295
7.5.4	JpaItemWriter	245	8.7.3	执行顺序	295
7.6	写 JMS 队列	248	8.7.4	Annotation	296
7.6.1	JmsItemWriter	248	8.7.5	属性 Merge	297
7.7	组合写	252			

第3篇 高级篇

第9章 作业流 Step Flow	300	10.2.3 重试拦截器	343
9.1 顺序 Flow	300	10.2.4 重试模板	345
9.2 条件 Flow	302	10.3 重启 Restart	353
9.2.1 next	303	10.3.1 重启 Job	353
9.2.2 ExitStatus VS BatchStatus	306	10.3.2 启动次数限制	355
9.2.3 decision 条件	308	10.3.3 重启已完成的任务	355
9.3 并行 Flow	311	第11章 扩展 Job、并行处理	357
9.4 外部 Flow 定义	314	11.1 可扩展性	357
9.4.1 Flow	314	11.2 多线程 Step	358
9.4.2 FlowStep	317	11.2.1 配置多线程 Step	359
9.4.3 JobStep	319	11.2.2 线程安全性	360
9.5 Step 数据共享	321	11.2.3 线程安全 Step	361
9.6 终止 Job	323	11.2.4 可重启的线程安全 Step	363
9.6.1 end	324	11.3 并行 Step	365
9.6.2 stop	326	11.4 远程 Step	366
9.6.3 fail	327	11.4.1 远程 Step 框架	366
第10章 健壮 Job	330	11.4.2 基于 SI 实现远程 Step	368
10.1 跳过 Skip	331	11.5 分区 Step	373
10.1.1 配置 Skip	331	11.5.1 关键接口	374
10.1.2 跳过策略 SkipPolicy	333	11.5.2 基本配置	376
10.1.3 跳过拦截器	335	11.5.3 文件分区	378
10.2 重试 Retry	338	11.5.4 数据库分区	382
10.2.1 配置 Retry	339	11.5.5 远程分区 Step	387
10.2.2 重试策略 RetryPolicy	341	后记	392

第1篇 入门篇

本篇从基本的入门讲起，通过介绍批处理、Spring Batch 基本特性、新特性，快速入门的 Hello World 等内容引领读者入门，进入批处理的世界。本篇包含两个章节。

第 1 章：向读者介绍什么是批处理，Spring Batch 框架适用的业务场景、技术场景，Spring Batch 的核心三层架构、Spring Batch 2.0、2.2 新特性，最后带领读者一起搭建 Spring Batch 的开发环境。

第 2 章：通过经典的 Hello World 示例向读者全面展示了 Spring Batch 的入门示例。

Spring Batch 简介

1.1 什么是批处理

现代互联网企业、金融业、电信业甚至传统行业通过 OLTP（联机事务处理）的业务系统积累了海量企业数据，需要企业应用能够在关键任务中进行批量处理来操作业务逻辑。通常情况下，此类业务并不需要人工参与就能够自动高效地进行复杂数据处理与分析。例如，定期对大批量数据进行业务处理（如银行对账和利率调整、或者跨系统的数据同步），或者是把从内部和外部系统中获取到的数据进行处理后集成到其他系统中去，这类工作被称之为“批处理”。

“批处理”工作在面对复杂的业务以及海量的数据处理时，无需人工干预，仅需定期读入批量数据，然后完成相应业务处理并进行归档操作。

典型的批处理应用有如下几个特点：

- （1）自动执行，根据系统设定的工作步骤自动完成；
- （2）数据量大，少则百万，多则千万甚至上亿；
- （3）定时执行，如每天执行、每周或每月执行。

从上面的描述中可以看出，批处理的整个流程可以明显地分为 3 个阶段：

- （1）读数据，数据可能来自文件、数据库或消息队列等；
- （2）处理数据，处理读取的数据并形成输出结果，如银行对账系统的资金对账处理；
- （3）写数据，将输出结果写入文件、数据库或消息队列等。

一个典型的批处理应用场景：系统 A 从数据库获取数据，经过业务处理后，导出系统 B 需要的数据到文件中，系统 B 读取该文件，经过业务处理后，最后存放在数据库中。通常情况下该动作在每天夜间 12:00~2:00 之间进行，此时对系统的性能影响最小。图 1-1 给出了典型批处理应用的场景。

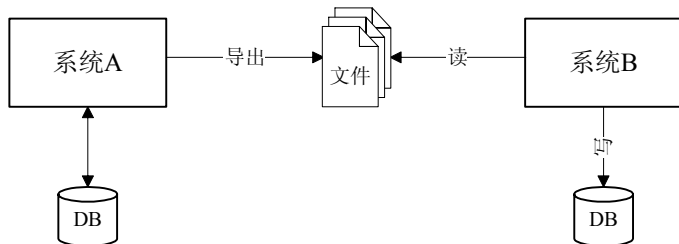


图 1-1 典型批处理应用场景

1.2 Spring Batch

Spring Batch 是一个轻量级的、完善的批处理框架，旨在帮助企业建立健壮、高效的批处理应用。Spring Batch 是 Spring 的一个子项目，使用 Java 语言并基于 Spring 框架为基础开发，使得已经使用 Spring 框架的开发者或者企业更容易访问和利用企业原有服务。

Spring Batch 提供了大量可重用的组件，包括日志、追踪、事务、任务作业统计、任务重启、跳过、重复、资源管理。对于大数据量和高性能的批处理任务，Spring Batch 同样提供了高级功能和特性来支持，比如分区功能、远程功能。总之，通过 Spring Batch 能够支持简单的、复杂的和大数据量的批处理作业。

Spring Batch 是一个批处理应用框架，不是调度框架，但需要和调度框架合作来构建完成批处理任务。它只关注批处理任务相关的问题，如事务、并发、监控、执行等，并不提供相应的调度功能。如果需要使用调用框架，在商业软件和开源软件中已经有很多优秀的企业级调度框架（如 Quartz、Tivoli、Control-M、Cron 等）可以使用。

1.2.1 典型场景

典型的批处理应用通常从数据库、文件或队列中读取数据，之后使用一些方法处理数据（抽取、分析、处理、过滤等），最终使用修改过的格式将数据写回目标系统。通常在一个无需用户交互的离线环境下，Spring Batch 能够自动进行基本的批处理迭代，也能够为一个数据集提供事务保证。批处理任务是一个大多数 IT 项目的组成部分，而 Spring Source 是唯一能够提供健壮的、企业级扩展的开源批处理框架。

Spring Batch 批处理框架支撑的业务场景：

- (1) 定期提交批处理任务；
- (2) 并行批处理，即并行处理任务；
- (3) 企业消息驱动处理；
- (4) 大规模的并行处理；
- (5) 手动或定时重启；
- (6) 按顺序处理依赖的任务（可扩展为工作流驱动的批处理）；
- (7) 部分处理，如在回滚时忽略记录；
- (8) 完整的批处理事务。

Spring Batch 批处理框架支撑的技术目标：

- (1) 利用 Spring 编程模型，使程序员专注于业务处理，让 Spring 框架管理流程；
- (2) 明确分离批处理的执行环境 and 应用；
- (3) 将通用核心的服务以接口形式提供；
- (4) 提供“开箱即用”的简单的默认的核心执行接口；
- (5) 提供 Spring 框架中配置、自定义和扩展服务；

- (6) 所有默认实现的核心服务能够容易地被扩展与替换，不会影响基础层；
- (7) 提供一个简单的部署模式，使用 Maven 进行编译。

1.2.2 Spring Batch 架构

Spring Batch 核心架构分为三层：应用层、核心层、基础架构层。具体参见图 1-2。

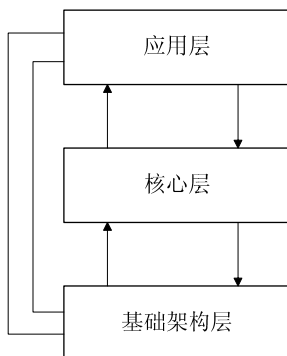


图 1-2 Spring Batch 三层核心架构

应用层包含所有的批处理作业，通过 Spring 框架管理程序员自定义的代码。核心层包含 Spring Batch 启动和控制所需要的核心类，如：JobLauncher、Job 和 step 等。应用层和核心层建立在基础架构层之上，基础架构层提供通用的读（ItemReader）、写（ItemWriter）和服务处理（如：RetryTemplate：重试模板；RepeatTemplate：重复模板，可以被应用层和核心层使用）。Spring Batch 的三层体系架构使得 Spring Batch 框架在不同的层级进行扩展，避免不同层级间的影响。

1.3 Spring Batch 优势

Spring Batch 是由 SpringSource 和 Accenture（埃森哲）合作开发的。Accenture 在批处理架构上有着丰富的工业级别的经验，贡献了之前专用的批处理体系框架（这些框架历经数十年研发和使用，为 Spring Batch 提供了大量的参考经验）；SpringSource 则有着深刻的技术认知和 Spring 框架编程模型。

Spring Batch 框架通过提供丰富的即开即用的组件和高可靠性、高扩展性的能力，使得开发批处理应用的人员专注于业务的处理，提升批处理应用的开发效率，通过 Spring Batch 可以快速地构建出轻量级的健壮的并行处理应用。

使用 Spring Batch 框架，你可以获得如下几小节所述的优势。

1.3.1 丰富的开箱即用组件

开箱即用组件包括对各种类型资源的读、写。

- 读：支持文本文件读、XML 文件读、数据库读、JMS 队列读。
- 写：支持写文本文件、XML 文件、数据库、JMS 队列。

该组件还提供作业仓库、作业调度器等基础设施，大大简化了批处理应用开发的复杂度。

1.3.2 面向 Chunk 的处理

面向 Chunk 的处理，支持多次读、一次写，避免了多次对资源的写入，大幅提升了批处理应用的处理效率。

1.3.3 事务管理能力

Spring Batch 框架默认采用 Spring 提供的声明式事务管理模型，面向 Chunk 的操作支持事务管理，同时支持为每个 tasklet 操作设置细粒度的事务配置：隔离级别、传播行为、超时设置等。

1.3.4 元数据管理

Spring Batch 框架自动记录 Job 的执行情况，包括 Job 的执行成功、失败、失败的异常信息，Step 的执行成功、失败、失败的异常信息，执行次数，重试次数，跳过次数，执行时间等，方便后期的维护和查看。

1.3.5 易监控的批处理应用

Spring Batch 框架提供多种监控技术，支持对批处理操作的信息进行查看和管理，通过 Spring Batch 框架为批处理应用提供了灵活的监控模式，包括：

- 直接查看数据库；
- 通过 Spring Batch 提供的 API 查看，基于 API，你可以打造自己的管理监控台；
- 通过 Spring Batch Admin 进行查看，Spring Batch Admin 是 Spring 的另外一个项目，通过该项目你可以通过 Web 控制台监控和操作 Job；
- 通过 JMX 控制台查看。

1.3.6 丰富的流程定义

Spring Batch 框架支持顺序任务、条件分支任务，基于顺序和条件任务可以组织复杂的任务流程。同时 Spring Batch 支持复用已经定义的 Job 或者 Step，同时提供 Job 和 Step 的继承

能力，方便任务的抽象。

1.3.7 健壮的批处理应用

Spring Batch 框架支持作业的跳过、重试、重启能力，避免因错误导致批处理作业的异常中断，例如如下操作。

- 跳过（Skip）：通常在发生非致命异常的情况下，应该不中断批处理应用；
- 重试（Retry）：发生瞬态异常情况下，应该能够通过重试操作避免该类异常，保证批处理应用的连续性和稳定性；
- 重启（Restart）：当批处理应用因错误发生错误后，应该能够在最后执行失败的地方重新启动 Job 实例。

1.3.8 易扩展的批处理应用

Spring Batch 框架通过并发和并行技术实现应用的横向、纵向扩展机制，满足数据处理性能的需要。

扩展机制包括：

- 多线程执行一个 Step（Multithreaded step）；
- 多线程并行执行多个 Step（Parallelizing step）；
- 远程执行作业（Remote chunking）；
- 分区执行（Partitioning Step）。

1.3.9 复用企业现有 IT 资产

Spring Batch 框架提供多种 Adapter 能力，使得企业现有的服务可以方便地集成到批处理应用中，避免了重新开发，达到复用企业遗留的服务资产。

1.4 Spring Batch 2.0 新特性

相对于 Spring Batch 1.X 系列，Spring Batch 2.X 系列提供了如下新的特性：

- （1）支持 Java 5；
- （2）非顺序的 Step 支持；
- （3）面向 Chunk 处理；
- （4）强化元数据访问；
- （5）增强扩展性；
- （6）可配置。

1.4.1 支持 Java 5

从 Spring Batch 2.X 版本开始，使用 Java 5 进行开发，支持 Java 5 提供的增强特性，如泛型、参数化类型等。

1.4.2 支持非顺序的 Step

Spring Batch 2.0 支持条件判断执行 Step 的方式。在 2.0 版本之前，仅支持顺序执行 Step。顺序执行 Step 参见图 1-3。

新的 Step 执行方式增加了条件判断功能，参见图 1-4。

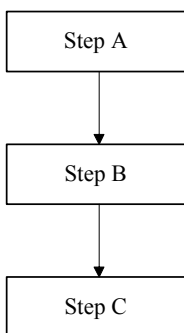


图 1-3 Spring Batch 1.X 版本仅支持顺序 Step

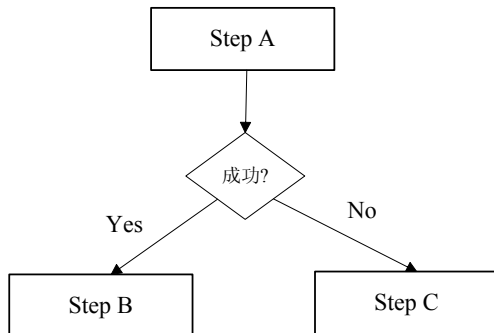


图 1-4 Spring Batch 2.X 版本支持条件 Step

代码清单 1-1 展示了如何配置图 1-4 中的条件流程。

代码清单 1-1 配置条件流程示例

```
1. <job id="job">
2.   <step id="stepA">
3.     <next on="FAILED" to="stepB" />
4.     <next on="*" to="stepC" />
5.   </step>
6.   <step id="stepB" next="stepC" />
7.   <step id="stepC" />
8. </job>
```

1.4.3 面向 Chunk 处理

Spring Batch 1.X 版本对数据处理默认提供的策略是面向 Item 处理。其序列图参见图 1-5。

在面向 Item 处理中，ItemReader 会返回一个对象（即 Item）给 ItemWriter 进行处理，Item 的数量为提交间隔的要求时提交计算结果。例如，如果提交所要求的 Item 数量为 3 时，ItemReader 和 ItemWriter 分别会被调用 3 次。使用代码清单 1-2 表示图 1-5 的执行。

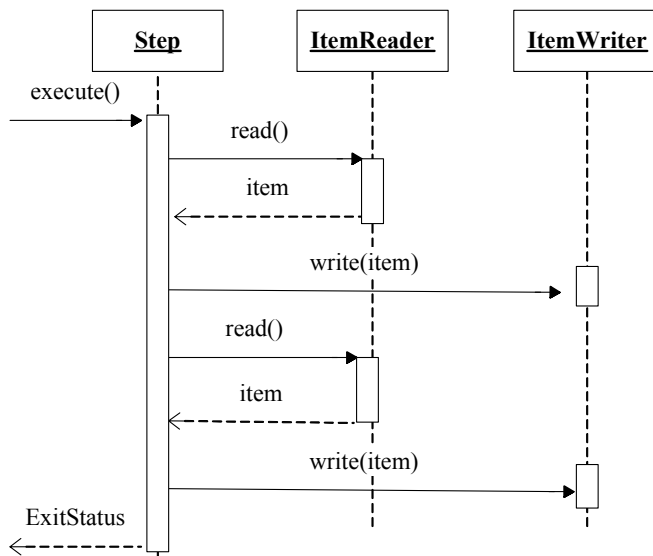


图 1-5 Spring Batch 1.X 版本支持面向 Item 的处理之序列图

代码清单 1-2 展示面向 Item 的处理伪代码

```

1. for(int i = 0; i < commitInterval; i++){
2.     Object item = itemReader.read();
3.     itemWriter.write(item);
4. }
  
```

对应的 `ItemReader` 和 `ItemWriter` 为了实现回滚的场景，需要在内部定义复杂的方法（如 `mark` 标记方法、`reset` 恢复方法、`clear` 清除方法等）。接口 `ItemReader` 和 `ItemWriter` 在 Spring Batch 1.X 版本的实现参见代码清单 1-3。

代码清单 1-3 Spring Batch 1.X 版本中 `ItemReader`、`ItemWriter` 的接口声明

```

1. public interface ItemReader {
2.     Object read() throws Exception;
3.     void mark() throws MarkFailedException;
4.     void reset() throws ResetFailedException;
5. }
6. public interface ItemWriter {
7.     void write(Object item) throws Exception;
8.     void flush() throws FlushFailedException;
9.     void clear() throws ClearFailedException;
10. }
  
```

由于处理的范围是一个 `Item`，如果要支持回滚场景就需要额外的方法，此时 `mark`、`reset`、`flush` 和 `clear` 就派上了用场。例如，在成功读/写了 2 个 `item` 之后，在写第三个 `item` 时发生了错误，整个事务就需要回滚，`writer` 中的 `clear` 方法会被调用，用于清空缓存，`Itemreader` 中的

reset 被调用，用于把 mark 方法所指向的数据游标复原。

Spring Batch 2.0 中支持面向 Chunk 的操作，简化了 ItemReader 和 ItemWriter 接口的复杂度。面向 Chunk 的操作序列图参见图 1-6。

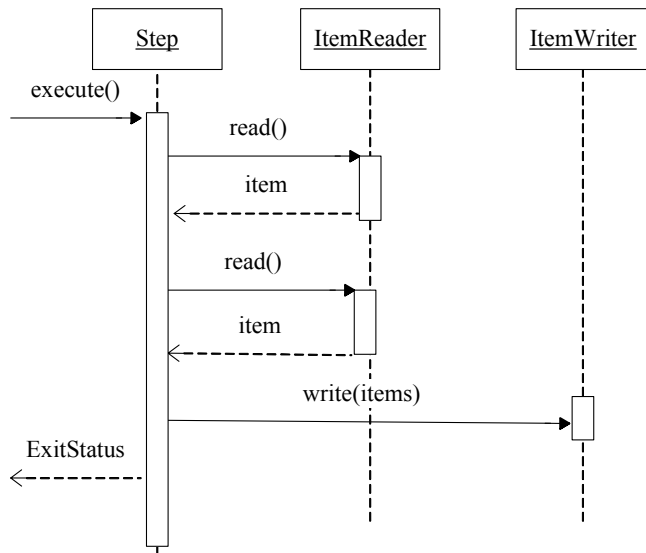


图 1-6 Spring Batch 2.X 版本支持面向 Chunk 的操作之序列图

按照面向 Chunk 的操作，如果提交间隔是 3 次，那么读操作被调用 3 次，写操作被调用 1 次。读 Item 被汇总到列表中，最终被统一写出。使用代码清单 1-4 来表示图 1-6 的执行：

代码清单 1-4 展示面向 Chunk 的处理伪代码

```
1. List items = new ArrayList();
2. for(int i = 0; i < commitInterval; i++){
3.     items.add(itemReader.read());
4. }
5. itemWriter.write(items);
```

面向 Chunk 的方案不仅更加简单更有扩展性，同时也让 ItemReader 和 ItemWriter 接口更加简洁。接口 ItemReader 和 ItemWriter 在 Spring Batch 2.X 版本的实现参见代码清单 1-5。

代码清单 1-5 Spring Batch 2.X 版本中 ItemReader、ItemWriter 的接口声明

```
1. public interface ItemReader<T> {
2.     T read() throws Exception, UnexpectedInputException, ParseException,
3.         NonTransientResourceException;
4. }
5. public interface ItemWriter<T> {
6.     void write(List<? extends T> items) throws Exception;
7. }
```

如代码所示，ItemReader 和 ItemWriter 接口不再包含 mark，reset，flush 和 clear 方法，使得读和写对象的创建更加直接。ItemReader 例子中，接口非常简单，框架会为开发者把读取的 item 缓存起来，以防 rollback 情况的发生。ItemWriter 也很简单，不再一次一个 item 地拿取，而是一次把整个 item 块都拿到，在把控制权交还给 step 前决定资源（如文件、数据库或者消息队列）的写入。

1.4.3.1 增加 ItemProcessor

在 Spring Batch 2.0 之前，Step 只依赖 ItemReader 和 ItemWriter，在 Spring Batch 2.0 中引入了 ItemProcessor（负责业务数据的处理）。

Spring Batch 1.X 的 Step 结构参见图 1-7。

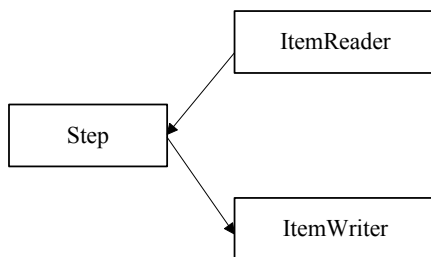


图 1-7 Spring Batch 1.X 的 Step 结构

通常的业务场景需要在数据写入之前，对数据进行处理，在 Spring Batch 1.X 版本中，可以使用组合模式，通过在读/写之间加入 ItemTransformer 这一层来实现。增加 ItemTransformer 之后的架构图参见图 1-8。

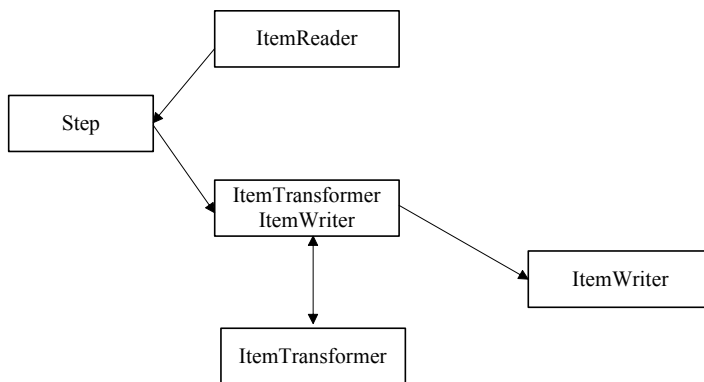


图 1-8 Spring Batch 1.X 通过 ItemTransformer 实现数据处理

Spring Batch 2.0 版本之后的 Step，将 ItemTransformer 重新命名为 ItemProcessor，和 ItemReader 与 ItemWriter 提升为相同的层级。Spring Batch 2.0 中增加 ItemProcessor 的架构图参见图 1-9。

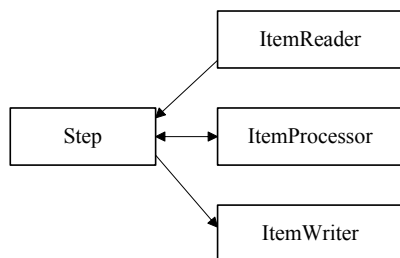


图 1-9 Spring Batch 2.0 通过 ItemProcessor 实现数据处理

1.4.4 元数据访问

Spring Batch 1.X 提供了元数据的增删查改接口 `JobRepository`。在 Spring Batch 2.X 版本中，新增元数据访问接口 `JobExplorer` 和 `JobOperator`。元数据访问接口的关系参见图 1-10。

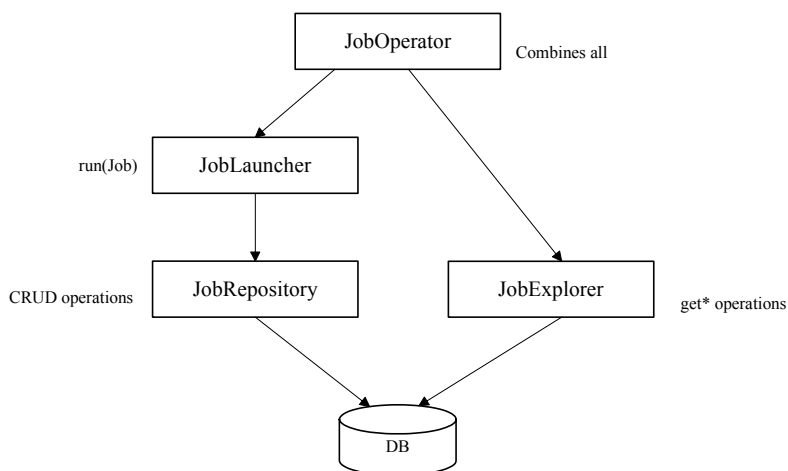


图 1-10 Spring Batch 2.X 中元数据管理接口关系图

1.4.5 扩展性

Spring Batch 1.X 只能在单个 JVM 中运行，支持任务在单个 JVM 多个线程并发执行。在 Spring Batch 2.0 增加支持多进程执行任务，包括远程分块和分区两个新功能。

远程分块

远程分块是一个把 `Step` 进行技术分割的工作，它不需要对处理数据的结构有明确了解。任何输入源能够使用单进程读取并在动态分割后作为“块”发送给远程的工作进程。远程进程实现了监听者模式，反馈请求、处理数据最终将处理结果异步返回。请求和返回之间的传输会被确保在发送者和单个消费者之间。Spring Batch 在 Spring Integration 顶部实现了远程分

块的特性。

分区

分区是另一个可选方案，它需要对数据的结构有一定的了解，如主键的范围、待处理的文件的名称等。这种模式的优点在于分区中每一个元素的处理器都能够像一个普通 Spring Batch 任务的单步一样运行，也不必去实现任何特殊的或是新的模式，来让它们能够更容易配置与测试。分区理论上比远程分块更有扩展性，因为分区并不存在从一个地方读取所有输入数据并进行序列化的瓶颈。

在 Spring Batch 2.0 中有两个接口支持分区：PartitionHandler 和 StepExecutionSplitter。PartitionHandler 知道执行结构——它需要将请求发送到远程步骤，并使用任何可以使用的网格或是远程技术收集计算结果。PartitionHandler 是一个 SPI，它和 Spring Batch 通过 TaskExecutor 为本地执行提供了一个外部实现方式。在需要进行有大量 I/O 操作的并发处理时，这个功能是很有用的。

1.4.6 可配置性

Spring Batch 1.X 没有独立的命名空间，所有批处理的配置需要作为 Spring 的配置项；Spring Batch 2.X 版本中增加了批处理的命名空间，简化了配置操作，对应的 XSD 可以通过 <http://www.springframework.org/schema/batch/spring-batch-2.2.xsd> 访问获取。

Spring Batch 1.X 中批处理配置文件参见代码清单 1-6。

代码清单 1-6 Spring Batch 1.X 中批处理配置文件示例代码

```
1. <bean id="footballJob"
2.     class="org.springframework.batch.core.job.SimpleJob">
3.     <property name="steps">
4.         <list>
5.             <!-- Step bean details omitted for clarity -->
6.             <bean id="playerload"/>
7.             <bean id="gameLoad"/>
8.             <bean id="playerSummarization"/>
9.         </list>
10.    </property>
11.    <property name="jobRepository" ref="jobRepository" />
12. </bean>
```

Spring Batch 2.X 中批处理配置文件参见代码清单 1-7。

代码清单 1-7 Spring Batch 2.X 中批处理配置文件示例代码

```
1. <job id="footballJob">
2.     <!-- Step bean details omitted for clarity -->
3.     <step id="playerload" next="gameLoad"/>
```

```

4.     <step id="gameLoad" next="playerSummarization"/>
5.     <step id="playerSummarization"/>
6. </job>

```

1.5 Spring Batch 2.2 新特性

相对于 Spring Batch 2.0 系列，Spring Batch 2.2.X 系列提供了如下新的特性：

- (1) 支持 Spring Data 集成；
- (2) 支持 Java 配置；
- (3) 重试模块（Spring Retry）重构；
- (4) 作业参数变化（Job Parameters）。

1.5.1 Spring Data 集成

Spring Batch 2.2 版本提供了对 Spring Data 的支持。Spring Data 是一个用于简化数据库访问，并支持云服务的开源框架。其主要目标是使得数据库的访问变得方便快捷，并支持 map-reduce 框架和云计算数据服务。此外，它还支持基于关系型数据库的数据服务，如 Oracle RAC 等。对于拥有海量数据的项目，可以用 Spring Data 来简化项目的开发，就如 Spring Framework 对 JDBC、ORM 的支持一样，Spring Data 会让数据的访问变得更加方便。

Spring Data 提供了对 NoSQL 类型的数据库提供了支持，Spring Batch 框架新增对 NoSQL 类型的数据库 MongoDB、Neo4j、Gemfire 的支持。

1.5.2 支持 Java 配置

Spring Batch 2.2 之前的版本提供了基于 XML 的方式配置 Job 作业，Spring Batch 2.2 之后的版本新增了 Java 方式的配置。为 Java 配置框架提供一组 annotation（@EnableBatchProcessing）和作业、作业步的工厂类（JobBuilderFactory、StepBuilderFactory）。

Spring Batch 2.2 之前版本提供的基于 XML 方式配置参见代码清单 1-8。

代码清单 1-8 Spring Batch 2.2 之前提供的基于 XML 方式定义作业

```

1. <batch>
2.     <job-repository/>
3.     <job id="myJob">
4.         <step id="step1".../>
5.         <step id="step2".../>
6.     </job>
7.     <beans:bean id="transactionManager".../>
8.     <beans:bean id="jobLauncher"
9.         class="org.springframework.batch.core.launch.support.

```

```

        SimpleJobLauncher">
10.     <beans:property name="jobRepository" ref="jobRepository"/>
11.     </beans:bean>
12. </batch>

```

Spring Batch 2.2 之后版本提供的基于 Java 配置方式参见代码清单 1-9。

代码清单 1-9 Spring Batch 2.2 之后提供的基于 Java 配置方式定义作业

```

1.  @Configuration
2.  @EnableBatchProcessing
3.  @Import(DataSourceConfiguration.class)
4.  public class AppConfig {
5.      @Autowired
6.      private JobBuilderFactory jobs;
7.
8.      @Bean
9.      public Job job() {
10.         return jobs.get("myJob").start(step1()).next(step2()).build();
11.     }
12.
13.     @Bean
14.     protected Step step1() {
15.         ...
16.     }
17.
18.     @Bean
19.     protected Step step2() {
20.         ...
21.     }
22. }

```

@EnableBatchProcessing 会依赖默认的 Spring Bean 对象, 包括 JobRepository (作业仓库)、JobLauncher (作业调度器)、JobRegistry (作业注册器)、PlatformTransactionManager (事务管理器)、JobBuilderFactory (作业构建工厂)、StepBuilderFactory (作业步构建工厂)。

1.5.3 Spring Retry

Spring Batch 提供的重试功能已经被抽取出来作为 Spring 一个独立的组件 Spring Retry 发布。重试组件在 Spring Batch 2.2 之前的包为: org.springframework.batch.retry; 在 Spring Batch 2.2 的版本重构为 org.springframework.retry。

1.5.4 Job Parameters

在 Spring Batch 2.2 之前的版本, 作业参数 (Job Parameters) 被强制用来标识作业实例 (Job

Instance)；在 Spring Batch 2.2 版本之后作业参数是否标识作业实例用户可以自由选择。

Spring Batch 2.2 之前的版本强制作业参数作为作业的实例，导致在作业重启的情况下无法修改作业参数；Spring Batch 2.2 之后的版本修改了这个缺陷。为此 Spring Batch 修改了自己的数据模型，在 Spring Batch 2.2 之前的版本作业参数与作业实例 ID 进行关联；Spring Batch 2.2 之后的版本将作业参数与作业执行器的 ID 关联。

1.6 开发环境搭建

Spring 批处理框架 1.0.0 版本于 2008 年 3 月份推出，推出版本非常快，截止到目前 2014 年 10 月最新的 2.X 系列 Release 的 GA 版本是 2.2.7 版本；3.X 系列已经推出 3.0.1 的 GA 版本，3.1.0 的 SNAPSHOT 已经推出。本书写作时使用的是 2.2.1 的 GA 版本，本书的所有源代码都是在 2.2.1 版本上调试通过的。本书配套源代码可以在 <https://github.com/jxtaliu/SpringBatchSample.git> 免费下载。

如果你是一个新的开发者，请到 Spring Batch 官网 <http://projects.spring.io/spring-batch/> 了解一下 Spring Batch 框架的内容。具体的下载地址为：<http://static.springframework.org/downloads/nightly/release-download.php?project=BATCH>；请下载最新的 RELEASE 2.2.1 版本（**BATCH/spring-batch-2.2.1.RELEASE-no-dependencies.zip**）。

下载后的文件为 `spring-batch-2.2.1.RELEASE-no-dependencies.zip`，请解压缩到目录 `spring-batch-2.2.1.RELEASE` 中，图 1-11 展示了解压缩后的目录结构。

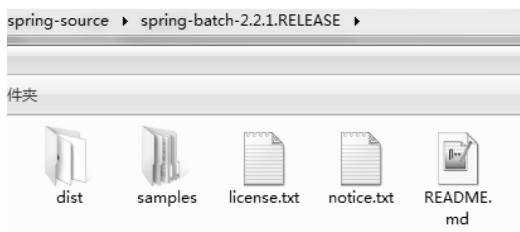


图 1-11 Spring Batch 2.2.1 解压缩后的目录结构

其中，

- `dist`: 编译好的 jar 和源文件压缩包。
- `samples`: 示例工程，默认是 maven 构件项目。
- `license.txt`: 授权的 license 文件，采用 Apache License。
- `notice.txt`: 使用 Spring Batch 的注意事项。
- `README.md`: 说明文档，包含如何获取源代码、编译及开发工具。

注意：为了编译 `samples` 项目，你需要安装 Maven，如果你还没有使用过 Maven 构建项目，请通过 Google 学习。Maven 的官方网站地址为：<http://maven.apache.org/>。

接下来的章节，我们通过“Hello World”示例展示如何快速使用 Spring Batch 框架。

Spring Batch 之 Hello World

本章通过一个实际信用卡账单对账例子，详细描述 Spring 批处理框架的使用方法，通过该例子，读者可以学会使用 Spring Batch 已经提供好的功能组件和基础设施，快速搭建批处理应用。

2.1 场景说明

个人使用信用卡消费，银行定期发送银行卡消费账单，本例模拟银行处理个人信用卡消费对账单对账，银行需要定期地把个人消费的记录导出成 CSV 格式文件，然后交给对账系统处理，本例子模拟银行读入 CSV 文件，经过处理后，生成新的对账单。示例场景架构参见图 2-1。

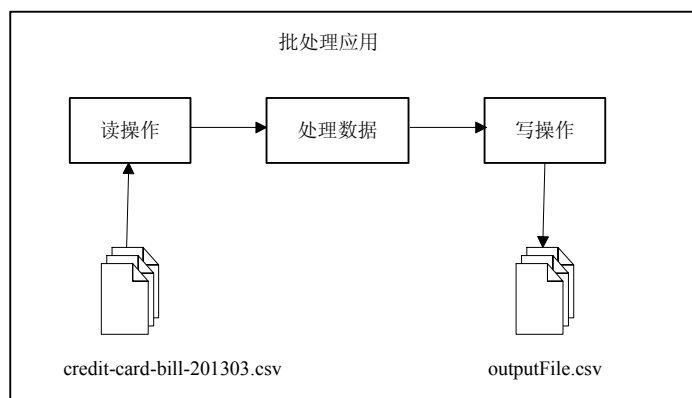


图 2-1 Hello World 信用卡消费对账单

2.2 项目准备

2.2.1 项目结构

项目的开发完成后的结构参见图 2-2。

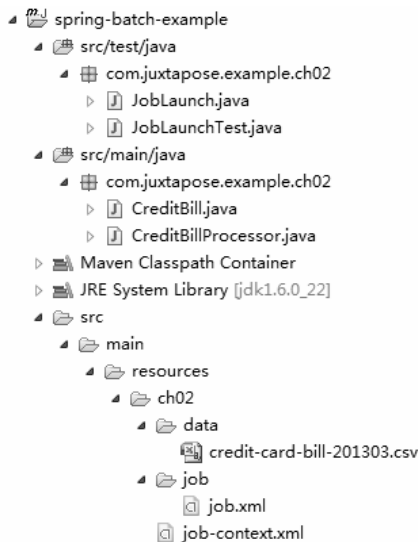


图 2-2 项目结构清单

文件说明如下。

- **CreditBill.java**: 表示信用卡消费记录领域对象。
- **CreditBillProcessor.java**: 记录处理类，本场景中没有做任何业务操作，仅打印账单信息。
- **JobLaunch.java**: 调用批处理作业类。
- **JobLaunchTest.java**: JUnit 单元测试类，使用 Spring 提供的测试框架类。
- **credit-card-bill-201303.csv**: 原始账单文件，存放账单消费条目。
- **job.xml**: 作业定义文件。
- **job-context.xml**: Spring Batch 批处理任务需要的基础信息。

2.2.2 准备对账单文件

在介绍代码之前，我们一起先看一下信用卡消费清单文件，该文件是 CSV（Comma Separated Value 的简写，通常都是纯文本文件）格式文件。credit-card-bill-201303.csv 的文件内容参见代码清单 2-1。

代码清单 2-1 credit-card-bill-201303.csv

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road
```

该对账单文件，每行表示一次信用卡的消费记录，中间用逗号分隔，分别表示银行卡账户、持卡人姓名、消费金额、消费日期、消费场所。

2.2.3 定义领域对象

为了映射上面的对账文件，需要开发信用卡账单领域对象 `CreditBill`，主要包括银行卡账户、持卡人姓名、消费金额、消费日期、消费场所等信息。该类的用途主要用于在 `ItemReader` 读取 CSV 文件中的数据后，转换为领域对象 `CreditBill`，供 `ItemProcessoe` 和 `ItemWriter` 使用。

领域对象 `CreditBill` 代码（此处省略了 `get*`，`set*` 方法，以节省篇幅）参见代码清单 2-2。

代码清单 2-2 领域对象 `CreditBill`

```
1. public class CreditBill {
2.     private String accountID = "";    /** 银行卡账户 ID */
3.     private String name = "";         /** 持卡人姓名 */
4.     private double amount = 0;        /** 消费金额 */
5.     private String date;              /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */
6.     private String address;           /** 消费场所 */
7. }
```

2.3 定义 job 基础设施

接下来介绍如何在 Spring 配置文件中定义批处理任务，首先介绍 `job-context.xml` 内容，`job-context.xml` 中定义了批处理任务中需要的基础设施，主要配置任务仓库、任务调度器、任务执行中用到的事务管理器。`job-context.xml` 内容参见代码清单 2-3。

代码清单 2-3 配置文件 `job-context.xml`

```
1. <bean id="jobRepository"
2.     class="org.springframework.batch.core.repository.support.
3.         MapJobRepositoryFactoryBean">
4. </bean>
5. <bean id="jobLauncher"
6.     class="org.springframework.batch.core.launch.support.
7.         SimpleJobLauncher">
8.     <property name="jobRepository" ref="jobRepository"/>
9. </bean>
10. <bean id="transactionManager"
11.     class="org.springframework.batch.support.transaction.
12.         ResourcelessTransactionManager"/>
```

该程序说明如下。

1~4 行：定义作业仓库，在 Spring Batch 框架中，任何任务的操作都会被记录在作业仓库中，Spring Batch 提供两种仓库，一种是内存的，另一种是数据库的。本例子中采用了内存

方式记录 Job 执行期产生的状态信息。

5~8 行：定义作业调度器，用来启动 Job；引用了 1-4 行定义的作业仓库。

9~11 行：定义了事务管理器，用于 Spring Batch 框架在对数据操作过程中提供事务能力。

2.4 定义对账 Job

job.xml 中定义批处理作业 billJob，billJob 共有一个 step，命名为 billStep，包含读、处理、写三个操作。job 配置内容参见代码清单 2-4。

代码清单 2-4 配置文件 job.xml

```
1.      <bean:import resource="classpath:ch02/job-context.xml"/>
2.      <job id="billJob">
3.          <step id="billStep">
4.              <tasklet transaction-manager="transactionManager">
5.                  <chunk reader="csvItemReader" writer="csvItemWriter"
6.                      processor="creditBillProcessor" commit-interval="2">
7.                  </chunk>
8.              </tasklet>
9.          </step>
10.     </job>
```

该程序说明如下。

1 行：导入 job-context.xml 文件，是 Spring 提供的功能，可以在不同的文件中定义 Spring 的配置信息，然后通过 import 方式组织导入。

2 行：定义名字为 billJob 的作业，该作业由一个名为 billStep 的 step 组成。

3~9 行：定义名字为 billStep 的作业步，billStep 作业步由一个面向批的操作组成。

4 行：定义批处理操作采用 job-context.xml 中定义的事务管理器，负责批处理中事务管理操作。

5~6 行：定义了面向批的操作，定义了读操作 csvItemReader、处理操作 creditBillProcessor、写操作 csvItemWriter；csvItemReader 负责从文件 credit-card-bill-201303.csv 中读取数据，creditBillProcessor 负责处理文件中的每一行数据，csvItemWriter 负责将 creditBillProcessor 处理的数据写到文件 outputFile.csv 中；commit-interval=2 表示提交间隔的大小，即每处理 2 条数据，进行一次写入操作，这样可以提高写的效率，在面向大数据量的批处理操作中，可以将 commit-interval 设置 1000~10000 的值。

2.4.1 配置 ItemReader

批处理操作的第一步是读取文本文件中的数据。csvItemReader 负责从文件中读取数据，并转换为信用卡对账单对象 CreditBill。csvItemReader 的配置内容参见代码清单 2-5。

代码清单 2-5 配置读数据 csvItemReader

```
1.      <!-- 读取信用卡账单文件,CSV 格式 -->
2.      <bean:bean id="csvItemReader"
3.          class="org.springframework.batch.item.file.FlatFileItemReader"
4.          scope="step">
5.          <bean:property name="resource"
6.              value="classpath:ch02/data/credit-card-bill-201303.csv"/>
7.          <bean:property name="lineMapper">
8.              <bean:bean
9.                  class="org.springframework.batch.item.file.mapping.
10.                     DefaultLineMapper">
11.                  <bean:property name="lineTokenizer" ref="lineTokenizer"/>
12.                  <bean:property name="fieldSetMapper">
13.                      <bean:bean class="org.springframework.batch.item.file.
14.                         mapping.BeanWrapperFieldSetMapper">
15.                          <bean:property name="prototypeBeanName" value=
16.                              "creditBill">
17.                              </bean:property>
18.                          </bean:bean>
19.                      </bean:property>
20.                  </bean:bean>
21.              </bean:property>
22.          </bean:bean>
23.      <!-- lineTokenizer -->
24.      <bean:bean id="lineTokenizer"
25.          class="org.springframework.batch.item.file.transform.
26.              DelimitedLineTokenizer">
27.          <bean:property name="delimiter" value=", "/>
28.          <bean:property name="names">
29.              <bean:list>
30.                  <bean:value>accountID</bean:value>
31.                  <bean:value>name</bean:value>
32.                  <bean:value>amount</bean:value>
33.                  <bean:value>date</bean:value>
34.                  <bean:value>address</bean:value>
35.              </bean:list>
36.          </bean:property>
37.      </bean:bean>
```

该程序说明如下。

2~3 行: csvItemReader 由 Spring Batch 提供的基础设施定义,此例使用 FlatFileItemReader 读文本文件; FlatFileItemReader 有两个属性: 一个是 resource, 表示需要读取的文件资源; 另

一个是 `lineMapper`，通过 `lineMapper` 可以把文本中的一行转换为领域对象 `Creditbill`。

5~6 行：读操作的文件，本例中读文件为 `classpath:ch02/data/credit-card-bill-201303.csv`。

7 行：定义 `lineMapper` 属性，本例中使用 Spring Batch 提供的基础设施 `org.springframework.batch.item.file.mapping.DefaultLineMapper`，`DefaultLineMapper` 有两个基本属性 `lineTokenizer` 和 `fieldSetMapper`。

10 行：定义 `lineTokenizer` 属性，`lineTokenizer` 定义文本中每行的分隔符号，以及每行映射成 `FieldSet` 对象后的 `name` 列表。具体定义参见 22~33 行。

11~17 行：定义 `fieldSetMapper` 属性，本例中使用 Spring Batch 提供的基础设施 `org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper`，`BeanWrapperFieldSetMapper` 根据 `lineTokenizer` 中定义的 `names` 属性映射到 `creditBill` 中，最终组装为信用卡账单对象 `CreditBill`。

22~33 行：定义 `lineTokenizer`，用于指定分隔符策略为“，”，并将分隔后的字段映射到属性 `names` 中定义的字段上，最终将字段映射成 `FieldSet` 对象中的 `name` 列表。

2.4.2 配置 ItemProcessor

批处理操作第二步是处理 `ItemReader` 中读取的数据。`creditBillProcessor` 负责处理 2.4.1 节中 `csvItemReader` 读入的数据，本例中的处理操作功能简单，仅仅打印了对账单信息，没有进行任何业务上的处理。`creditBillProcessor` 的配置内容参见代码清单 2-6。

代码清单 2-6 配置处理数据 `creditBillProcessor`

```
1.     <bean:bean id="creditBillProcessor" scope="step"
2.         class="com.juxtapose.example.ch02.CreditBillProcessor">
3.     </bean:bean>
```

`com.juxtapose.example.ch02.CreditBillProcessor` 处理类定义参见代码清单 2-7。`CreditBillProcessor` 实现了接口 `ItemProcessor`，方法 `process` 用于处理 `csvItemReader` 读取的数据（参数 `bill`），并将处理后的数据返回，返回的数据会被写操作 `ItemWriter` 处理。

代码清单 2-7 数据处理类 `CreditBillProcessor`

```
1. public class CreditBillProcessor implements
2.     ItemProcessor<CreditBill, CreditBill> {
3.
4.     public CreditBill process(CreditBill bill) throws Exception {
5.         System.out.println(bill.toString());
6.         return bill;
7.     }
8. }
```

2.4.3 配置 ItemWriter

批处理操作的第三步是将 `ItemProcessor` 中处理的数据写入给定的目标存储中。`csvItemWriter` 负责将 `ItemProcessor` 处理后的信用卡账单对象 `CreditBill` 写入到文本文件 `outputFile.csv` 中。

`csvItemWriter` 的配置信息参见代码清单 2-8。

代码清单 2-8 配置写数据 `csvItemWriter`

```
1.      <!-- 写信用卡账单文件，CSV 格式 -->
2.      <bean:bean id="csvItemWriter"
3.          class="org.springframework.batch.item.file.FlatFileItemWriter"
4.          scope="step">
5.          <bean:property name="resource" value="file:target/ch02/outputFile.
6.              csv"/>
7.          <bean:property name="lineAggregator">
8.              <bean:bean
9.                  class="org.springframework.batch.item.file.transform.
10.                      DelimitedLineAggregator">
11.                  <bean:property name="delimiter" value=","/></bean:property>
12.                  <bean:property name="fieldExtractor">
13.                      <bean:bean
14.                          class="org.springframework.batch.item.file.
15.                              transform.BeanWrapperFieldExtractor">
16.                              <bean:property name="names"
17.                                  value="accountID,name,amount,date,address">
18.                                  </bean:property>
19.                              </bean:bean>
20.                          </bean:property>
21.                      </bean:bean>
22.                  </bean:property>
23.              </bean:bean>
24.          </bean:property>
25.      </bean>
```

该程序说明如下。

2~3 行：`csvItemWriter` 由 Spring Batch 提供的基础设施定义，此例使用 `FlatFileItemWriter` 写入文本文件；`FlatFileItemWriter` 有两个属性：一个是 `resource`，表示需要写入的文件资源；另一个是 `lineAggregator`，`lineAggregator` 负责将信用卡账单对象根据定义的规则转换为一个文本。

5 行：定义 `resource` 属性，本例中写入的文件是：`file:target/ch02/outputFile.csv`。

6~21 行：定义 `lineAggregator` 属性，`lineAggregator` 默认使用了 Spring Batch 提供的基础设施类 `org.springframework.batch.item.file.transform.DelimitedLineAggregator`，`DelimitedLineAggregator` 共有两个属性：一个是 `delimiter`，另一个是 `fieldExtractor`。

10 行：表示写入的文本以逗号分隔。

11~19 行：表示将信用卡账单对象 Creditbill 根据提供的 names 属性定义的名字转换为 Object 数组，最终由 DelimitedLineAggregator 转化为一行以逗号分隔的文本。

2.5 执行 Job

本节提供两种方式执行批处理作业：Java 调用和 Junit 单元测试。

2.5.1 Java 调用

到此为止，Spring Batch 配置文件完成，ItemProcessor 和信用卡账单对象 CreditBill 也已经完成，下面描述如何执行定义任务 billJob。还记得上面我们提到的 jobLauncher 对象吗，我们可以利用 jobLauncher 对作业进行调度。通过 Java 调用批处理作业的代码参见代码清单 2-9。

代码清单 2-9 Java 调用批处理作业

```
1. ApplicationContext context = new ClassPathXmlApplicationContext(
    "ch02/job/job.xml");
2. JobLauncher launcher = (JobLauncher) context.getBean("jobLauncher");
3. Job job = (Job) context.getBean("billJob");
4. try {
5.     JobExecution result = launcher.run(job, new JobParameters());
6.     System.out.println(result.toString());
7. } catch (Exception e) {
8.     e.printStackTrace();
9. }
```

该程序说明如下。

1 行：初始化 Spring 上下文，传入任务定义文件 job.xml。

2 行：获取作业调度器，根据名字为"jobLauncher"的 Bean 从 Spring 上下文获取。

3 行：获取名字为"billJob"的任务对象。

5 行：通过 jobLauncher 的 run 方法执行 billJob 任务。

执行上面后，读者可以查看输出文件 outputFile.csv 内容，所有的信用卡消费记录已经全部写入到 outputFile.csv 文件中，具体内容参见代码清单 2-10。

代码清单 2-10 Java 调用写入后的 outputFile.csv 文件

```
1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road
```

读者仔细观察控制台，会有如下的信息在控制台输出，具体内容参见代码清单 2-11。

代码清单 2-11 Java 调用的控制台输出

```
accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
12:00:08;address=Lu Jia Zui road
accountID=4047390012345678;name=tom;amount=320.0;date=2013-2-3
10:35:21;address=Lu Jia Zui road
accountID=4047390012345678;name=tom;amount=674.7;date=2013-2-6
16:26:49;address=South Linyi road
accountID=4047390012345678;name=tom;amount=793.2;date=2013-2-9
15:15:37;address=Longyang road
accountID=4047390012345678;name=tom;amount=360.0;date=2013-2-11
11:12:38;address=Longyang road
accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28
20:34:19;address=Hunan road
JobExecution: id=0, version=2, startTime=Thu Feb 28 21:31:34 CST 2013,
endTime=Thu Feb 28 21:31:34 CST 2013, lastUpdated=Thu Feb 28 21:31:34 CST 2013,
status=COMPLETED, exitStatus=exitCode=COMPLETED;exitDescription=,
job=[JobInstance: id=0, version=0, JobParameters=[{}], Job=[billJob]]
```

2.5.2 JUnit 单元测试

JUnit 单元测试的一些代码如下。

代码清单 2-12 JUnit 调用批处理作业

```
1. @RunWith(SpringJUnit4ClassRunner.class)
2. @ContextConfiguration(locations={"/ch02/job/job.xml"})
3. public class JobLaunchTest {
4.     @Autowired
5.     private JobLauncher jobLauncher;
6.     @Autowired@Qualifier("billJob")
7.     private Job job;
8.
9.     @Before
10.    public void setUp() throws Exception {
11.    }
12.
13.    @After
14.    public void tearDown() throws Exception {
15.    }
16.
17.    @Test
18.    public void importProducts() throws Exception {
19.        JobExecution result = jobLauncher.run(job, new JobParameters());
20.        System.out.println(result.toString());
21.    }
22. }
```

该程序说明如下。

1 行: `@RunWith(SpringJUnit4ClassRunner.class)`表示该测试用例是运用 junit4 进行测试

2 行: `@ContextConfiguration(locations={"/ch02/job/job.xml"})`表示设置 Spring 上下文加载的文件路径, 这里加载"/ch01/job/job.xml"文件, 也可以写成 `classpath:xxx.xml` 的格式。

4~5 行: 使用 `@Autowired` 自动装配 `jobLauncher` 实例, Spring 上下文自动使用配置文件中定义的 bean id 为 "jobLauncher" 的对象。

6~7 行: 使用 `@Qualifier("billJob")` 手工指定装配对象的 bean id 值。

9~11 行: `@Before` 表示在单元测试用例执行前调用的方法。

13~15 行: `@After` 表示在单元测试用例执行后调用的方法。

17~20 行: `@Test` 表示该方法是单元测试用例, 本测试用例中使用 `jobLauncher` 调用批处理作业任务。

执行单元测试, 执行成功后的效果参见图 2-3。

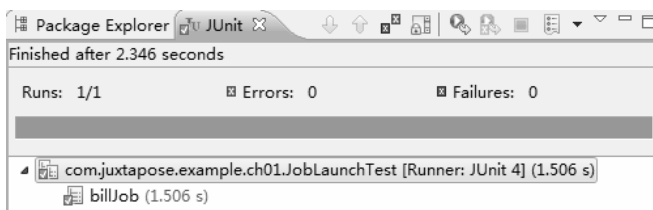


图 2-3 JUnit 执行批处理任务效果图

执行单元测试后, 读者可以查看输出文件 `outputFile.csv` 内容, 所有的信用卡消费记录已经全部写入到 `outputFile.csv` 文件中, 具体内容参见代码清单 2-13。

代码清单 2-13 JUnit 调用写入后的 `outputFile.csv` 文件

```
1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road
```

读者仔细观察控制台, 会有如下的信息在控制台输出, 具体内容参见代码清单 2-14。

代码清单 2-14 JUnit 调用的控制台输出

```
accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
12:00:08;address=Lu Jia Zui road
accountID=4047390012345678;name=tom;amount=320.0;date=2013-2-3
10:35:21;address=Lu Jia Zui road
accountID=4047390012345678;name=tom;amount=674.7;date=2013-2-6
16:26:49;address=South Linyi road
accountID=4047390012345678;name=tom;amount=793.2;date=2013-2-9
```

```
15:15:37;address=Longyang road
accountID=4047390012345678;name=tom;amount=360.0;date=2013-2-11
11:12:38;address=Longyang road
accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28
20:34:19;address=Hunan road
JobExecution: id=0, version=2, startTime=Thu Feb 28 21:31:34 CST 2013,
endTime=Thu Feb 28 21:31:34 CST 2013, lastUpdated=Thu Feb 28 21:31:34 CST 2013,
status=COMPLETED, exitStatus=exitCode=COMPLETED;exitDescription=,
job=[JobInstance: id=0, version=0, JobParameters=[{}], Job=[billJob]]
```

2.6 概念预览

基于 Spring Batch 框架可以快速配置出复杂的批处理应用。我们和读者一起总结一下截止目前我们提到的 Spring Batch 中的一些基本概念。

Job Repository: 作业仓库，负责 Job、Step 执行过程中的状态保存。

Job launcher: 作业调度器，提供执行 Job 的入口。

Job: 作业，由多个 Step 组成，封装整个批处理操作。

Step: 作业步，Job 的一个执行环节，由多个或者一个 Step 组装成 Job。

Tasklet: Step 中具体执行逻辑的操作，可以重复执行，可以设置具体的同步、异步操作等。

Chunk: 给定数量 Item 的集合，可以定义对 Chunk 的读操作、处理操作、写操作，提交间隔等，这是 Spring Batch 框架的一个重要特性。

Item: 一条数据记录。

ItemReader: 从数据源（文件系统、数据库、队列等）中读取 Item。

ItemProcessor: 在 Item 写入数据源之前，对数据进行处理（如：数据清洗，数据转换，数据过滤，数据校验等）。

ItemWriter: 将 Item 批量写入数据源（文件系统、数据库、队列等）。

本节我们对这些概念有个基本了解，下面的章节我们将详细描述这些概念。

第 2 篇 基本篇

本篇重点讲述了批处理的核心概念、典型的作业配置、作业步配置，以及 Spring Batch 框架中经典的三步走策略：数据读、数据处理、数据写。详细地介绍了如何对分隔符类型文件、定长类型文件、JSON 格式文件、复杂类型格式文件、XML 文件、数据库、JMS 消息队列中的数据进行读操作、处理、写操作，对于数据库的操作详细介绍了使用 JDBC、Hibernate、存储过程、JPA、Ibatis 等处理。本篇包含六个章节。

第 3 章：介绍 Spring Batch 的基本概念，主要包括命名空间、作业 Job、作业步 Step、执行上下文 Execution Context、作业仓库 Job Repository、作业调度器 Job Launcher、条目读 Item Reader、条目处理 Item Processor、条目写 Item Writer 等。

第 4 章：带领读者配置作业 Job，包括作业的重启、作业拦截器、作业参数校验、抽象作业、作业继承、作业步 Scope、属性后绑定技术，最后向读者介绍了如何在定时任务、命令行、Web 应用中执行作业 Job、停止作业 Job。

第 5 章：带领读者配置作业步 Step，包括作业步抽象、继承、拦截器、重启、提交间隔、异常跳过、重试、完成策略，最后向读者介绍了作业步的事物和拦截器。

第 6 章：介绍读数据 ItemReader，主要包括对 CSV 格式文件、XML 格式文件、JSON 格式文件、多文件、DB、JMS 队列等各种资源的读取组件，最后向读者介绍了服务复用、自定义 ItemReader、读拦截器等功能。

第 7 章：介绍写数据 ItemWriter，主要包括对 CSV 格式文件、XML 格式文件、JSON 格式文件、多文件、DB、JMS 队列、发送邮件等各种资源的写组件，最后向读者介绍了服务复用、自定义 ItemWriter、写拦截器等功能。

第 8 章：介绍数据处理 ItemProcessor，主要包括数据转换、数据过滤、数据校验、组合数据处理，最后向读者介绍了服务复用、处理拦截器。

Spring Batch 基本概念

Spring Batch 框架采用了埃森哲提供的经典的批处理框架模型，该框架模型在包括 COBOL、C++、C#、Java 等经典平台得到十多年的验证。图 3-1 是该经典批处理框架的架构图。

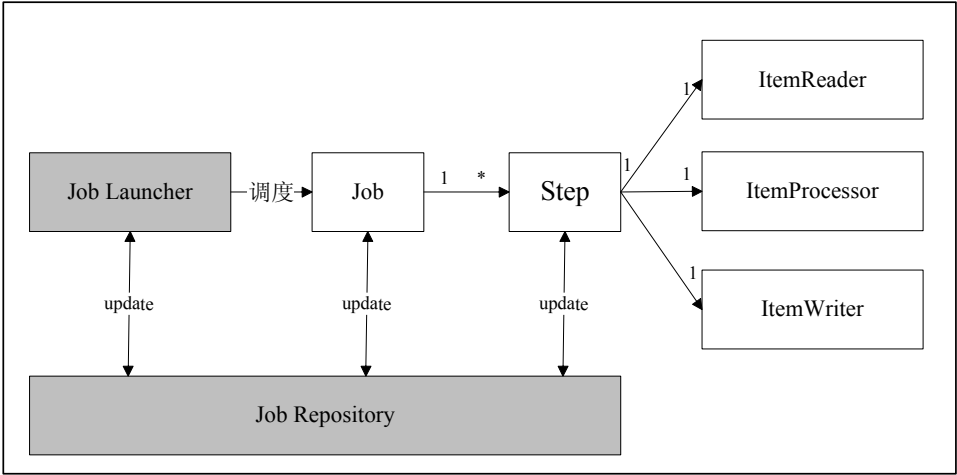


图 3-1 Spring Batch 批处理框架架构图

每个作业 Job 有 1 个或者多个作业步 Step; 每个 Step 对应一个 ItemReader、ItemProcessor、ItemWriter; 通过 Job Launcher 可以启动 Job, 启动 Job 时需要从 JobRepository 获取存在的 Job Execution; 当前运行的 Job 及 Step 的结果及状态会保存在 JobRepository 中。

表 3-1 列出了 Spring Batch 中用到的主要领域对象。

表 3-1 Spring Batch 主要领域对象

领域对象	描 述
Job	作业。批处理中的核心概念，是 Batch 操作的基础单元
Job Instance	作业实例。每个作业执行时，都会生成一个实例，实例会被存放在 JobRepository 中，如果作业失败，下次重新执行该作业的时候，会使用同一个作业实例；对于 Job 和 Job Instance 的关系，大家可以想象为 Java 类定义和 Java 对象实例的关系
Job Parameters	作业参数。它是一组用来启动批处理任务的参数，在启动 Job 时候，可以设置任何需要的作业参数，需要注意作业参数会用来标识作业实例，即不同的 Job 实例是通过 Job 参数来区分的

续表

领域对象	描 述
Job Execution	作业执行器。其负责具体 Job 的执行，每次运行 Job 都会启动一个新的 Job 执行器
Job Repository	作业仓库。其负责存储作业执行过程中的状态数据及结果，为 JobLauncher、Job、Step 提供标准的 CRUD 实现
Job Launcher	作业调度器。它根据给定的 JobParameters 执行作业
Step	作业步。Job 的一个执行环节，多个或者一个 Step 组装成 Job，封装了批处理任务中的一个独立的连续阶段
Step Execution	作业步执行器。它负责具体 Step 的执行，每次运行 Step 都会启动一个新的执行器
Tasklet	Tasklet。Step 中具体执行逻辑的操作，可以重复执行，可以设置具体的同步、异步操作等
Execution Context	执行上下文。它是一组框架持久化与控制的 key/value 对，能够让开发者在 Step Execution 或 Job Execution 范畴保存需要进行持久化的状态
Item	条目。一条数据记录
Chunk	Item 集合。它给定数量 Item 的集合，可以定义对 Chunk 的读操作、处理操作、写操作，提交间隔等
Item Reader	条目读。其表示 step 读取数据，一次读取一条
Item Processor	条目处理。用于表示 item 的业务处理
Item Writer	条目写。用于表示 step 输出数据，一次输出一批

本章为了解释清楚概念，会继续使用 HelloWorld 章节的示例。在原有示例的基础上，做了部分调整，具体代码参见第 3 章中的内容。

3.1 命名空间

Spring Batch 2.X 版本中增加了批处理的命名空间，简化了配置操作，对应的 XSD 可以通过 <http://www.springframework.org/schema/batch/spring-batch-2.2.xsd> 访问获取。通过引用命名空间，在进行 Job 配置的时候简化了配置文件的复杂度。

代码清单 3-1 给出了命名空间的配置信息。

代码清单 3-1 Spring Batch 2.X 提供的命名空间

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <bean:beans xmlns="http://www.springframework.org/schema/batch"
3.     xmlns:bean="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xmlns:p="http://www.springframework.org/schema/p"
6.     xmlns:tx="http://www.springframework.org/schema/tx"
7.     xmlns:aop="http://www.springframework.org/schema/aop">

```

```

8.      xmlns:context="http://www.springframework.org/schema/context"
9.      xsi:schemaLocation="http://www.springframework.org/schema/beans
10.     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
11.     http://www.springframework.org/schema/tx
12.     http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
13.     http://www.springframework.org/schema/aop
14.     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
15.     http://www.springframework.org/schema/context
16.     http://www.springframework.org/schema/context/spring-context-2.5.xsd
17.     http://www.springframework.org/schema/batch
18.     http://www.springframework.org/schema/batch/spring-batch-2.2.xsd">

```

具体代码参见源代码文件：/spring-batch-example/src/main/resources/ch03/job/job.xml。

3.2 Job

批处理作业 Job 由一组 Step 组成，同时是作业配置文件的顶层元素。每个作业有自己的名字、可以定义 Step 执行的顺序，以及定义作业是否可以重启。Job 的主要属性参见图 3-2。

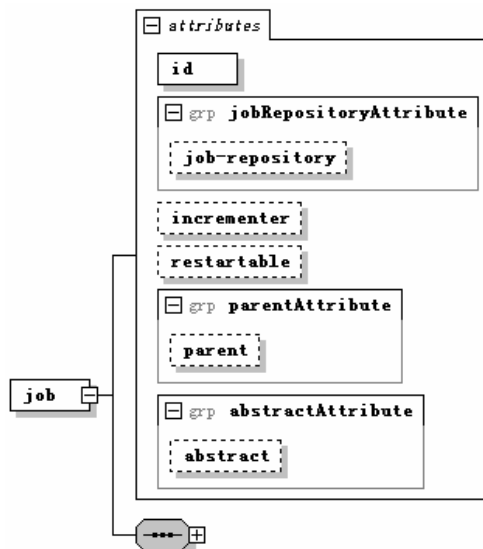


图 3-2 Job 元素的 Schema 定义

具体的属性定义参见第 4 章进行的描述。

Job 执行的时候会生成一个 Job Instance（作业实例），Job Instance 包含执行 Job 期间产生的数据以及 Job 执行的状态信息等；Job Instance 通过 Job Name（作业名称）和 Job Parameter（作业参数）来区分；每次 Job 执行的时候都有一个 Job Execution（作业执行器），Job Execution 负责具体 job 的执行。Job、Job Instance 和 Job Execution 三者的关系参见图 3-3。

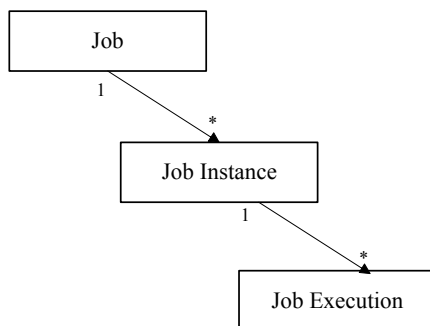


图 3-3 Job、Job Instance 和 Job Execution 三者关系图

一个 Job 可能有一到多个 Job Instance。

一个 Job Instance 可能有一到多个 Job Execution。

3.2.1 Job Instance

Job Instance（作业实例）是一个运行期的概念，Job 每执行一次都会涉及一个 Job Instance。Job Instance 来源可能有两种：一种是根据设置的 Job Parameters 从 Job Repository（作业仓库）中获取一个；如果根据 Job Parameters 从 Job Repository 没有获取 Job Instance，则新创建一个新的 Job Instance。

为了更好地了解 Job Instance，我们仍然使用第 2 章中用到的对账单的 Job。这里和第 2 章的变化在于将 Job Repository 由内存转到存放在数据库中，具体的配置信息请参见 3.5.3 章节的配置。

说明：在执行下面的示例前，需要根据 Spring Batch 框架提供的数据库脚本完成数据库的初始化，数据库脚本位置存放在 `spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core` 目录下，针对不同的数据库提供了不同的脚本，在运行本书的示例代码前，需要初始化 `schema-mysql.sql` 文件。

在执行对账单 Job 的时候，增加了参数配置信息，具体参见代码清单 3-2。

代码清单 3-2 以参数方式执行 Job

```

1. public static void executeJob(String jobPath, String jobName,
2.                               JobParametersBuilder builder) {
3.     ApplicationContext context = new ClassPathXmlApplicationContext
4.         (jobPath);
5.     JobLauncher launcher = (JobLauncher) context.getBean("jobLauncher");
6.     Job job = (Job) context.getBean(jobName);
7.     try {
8.         JobExecution result = launcher.run(job, builder.toJobParameters());
9.         System.out.println(result.toString());
10.    } catch (Exception e) {

```



```

10.         e.printStackTrace();
11.     }
12. }
13.
14. /**
15.  * @param args
16.  */
17. public static void main(String[] args) {
18.     executeJob("ch03/job/job.xml", "billJob",
19.         new JobParametersBuilder().addString("date", "20130308"));
20. }

```

该程序说明如下。

7 行：调用 Job 的时候增加了 JobParameters，根据配置的 JobParameters 可以唯一地区分不同的 Job Instance。

19 行：设置调用 Job 时候的参数名称为“date”，参数值为“20130308”，参数的数据类型为“String”类型

为了展示 Job Instance 和 Job Execution 及 Job 的关系，请读者按照下面的操作步骤执行示例代码：

- (1) 执行类 com.juxtapose.example.ch03.JobLaunch，第一次确保参数为“20130308”；
- (2) 将文件 ch03\data\credit-card-bill-201303-bad.csv 中的内容替换到文件 ch03\data\credit-card-bill-201303.csv 中，以参数“20130309”执行类 com.juxtapose.example.ch03.JobLaunch；
- (3) 恢复文件 ch03\data\credit-card-bill-201303.csv 内容，以参数“20130309”再次执行类 com.juxtapose.example.ch03.JobLaunch。

经过三次执行后，可以到数据库中看出，一共有两个 Job Instance 和三个 Job Execution。Job Instance 对应的数据库表为 BATCH_JOB_INSTANCE，Job Execution 对应的数据库表为 BATCH_JOB_EXECUTION。

图 3-4 展示了两个 Job Instance，编号分别是 1 和 2。

	JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
1	1	0	billJob	2de7ecd6c3ef4d75cf6ec6013f576c80
2	2	0	billJob	ee1c173b88cdb2f97654eac50bacc51d

图 3-4 Job Instance 所在表 BATCH_JOB_INSTANCE 的数据

图 3-5 展示了三个 Job Execution，编号分别是 1、2、3，对应的 Job Instance 分别是 1、2、2。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	STATUS	CREATE_TIME
1	1	2	1	COMPLETED	2013-03-07 21:46:18
2	2	2	2	FAILED	2013-03-07 21:47:01
3	3	2	2	COMPLETED	2013-03-07 21:47:29

图 3-5 Job Execution 所在表 BATCH_JOB_EXECUTION 的数据

执行三次产生的 Job Instance 和 Job Execution，参见图 3-6。

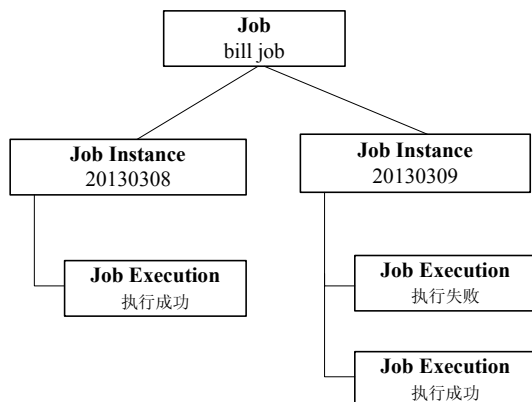


图 3-6 Job、Job Instance 和 Job Execution 三者示例图

以参数 `date=20130308` 执行了一次 `billJob`，执行成功；以参数 `date=20130309` 第一次执行 `billJob` 失败，重新以参数 `date=20130309` 第二次执行作业 `billJob`，执行成功。

Job Instance 对应的数据库表是：BATCH_JOB_INSTANCE。

具体表的字段信息参见 3.5.4.1 节。

Job Instance 对应的 Java 类是 `org.springframework.batch.core.JobInstance`。

小结：

- 第一次执行 Job 时候，会创建一个新的 Job Instance 和新的 Job Execution；
- 每次执行 Job 的时候，都会创建一个新的 Job Execution；
- 已经完成的 Job Instance，不能被重新执行；
- 在同一时刻，只能有一个 Job Execution 可以执行同一个 Job Instance。

3.2.2 Job Parameters

Job 通过 Job Parameters 来区分不同的 Job Instance。简单地说 Job Name + Job Parameters 来唯一地确定一个 Job Instance。如果 Job Name 一样，则 Job Parameters 肯定不一样；但是对于不同的 Job 来说，允许有相同的 Job Parameters。Job Instance、Job Name 和 Job Parameters 三个关系参见图 3-7。

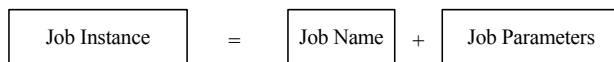


图 3-7 Job Instance、Job Name 和 Job Parameters 三个关系

3.2.1 节执行了 3 次 Job，共使用了两个不同的 Job Parameters，具体参见图 3-8。

Job Parameters 共支持四种类型的参数，数据类型分别是 String、Date、Long、Double。同时 Spring Batch 框架提供了 JobParametersBuilder 来构建参数，JobParametersBuilder 提供的构造参数参见图 3-9。

	JOB_EXECUTION_ID	TYPE_CD	KEY_NAME	STRING_VAL
1		1	STRING	date
2		2	STRING	date
3		3	STRING	date

图 3-8 Job Parameter 所在表 BATCH_JOB_EXECUTION_PARAMS 的数据

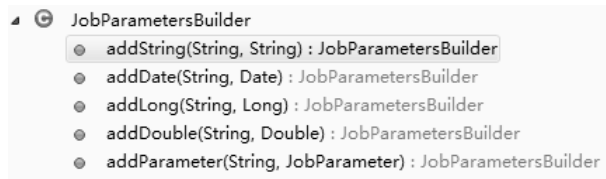


图 3-9 JobParametersBuilder 提供构建 Job Parameters 的操作

JobParameters 对应的数据库表是：BATCH_JOB_EXECUTION_PARAMS，表信息的具体描述参见 3.5.4.2 节。

JobParameters 对应的 Java 类是：org.springframework.batch.core.JobParameters。

Job Name 和相同的参数一样的情况下执行 Job，会发生错误。3.2.1 节中以参数为“20130308”的 billJob 已经执行过一次且成功，下面以参数“20130308”再次执行 billJob。操作步骤如下：

- (1) 修改类 com.juxtapose.example.ch03.JobLaunch 中的参数值为“20130308”；
- (2) 运行类 com.juxtapose.example.ch03.JobLaunch。

控制台会出现代码清单 3-3 的错误。

代码清单 3-3 相同 Job Name、Job Parameters 执行 Job 错误信息

```
org.springframework.batch.core.repository.JobInstanceAlreadyCompleteException: A job instance already exists and is complete for parameters={date=20130308}. If you want to run this job again, change the parameters.
    at org.springframework.batch.core.repository.support.SimpleJobRepository.createJobExecution(SimpleJobRepository.java:122)
```

该错误信息表示已经完成的 Job Instance 不能被再次执行，因为传入的参数相同，第二次执行 billJob 没有创建新的 Job Instance，而是根据参数“20130308”从数据库中读出上次已经完成的 Job Instance 来执行，导致出现上面的错误。

3.2.3 Job Execution

Job Execution 表示 Job 执行的句柄，根据上面的描述可知，一次 Job 的执行可能成功也可能失败。只有 Job Execution 执行成功后，对应的 Job Instance 才会被完成。根据上面例子对 billJob 的执行，参数为“20130308”的任务一次执行成功；参数为“20130309”的任务第一次执行失败，第二次执行成功。参数为“20130309”的任务共执行两次，共有两个 Job Execution 被创建，但是它们都对对应同一个 Job Instance。

通过章节 3.2.1 中执行后，对应的 Job Execution 信息参见图 3-10。

表 BATCH_JOB_EXECUTION 中的数据如下。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	STATUS	CREATE_TIME
1	1	2	1	COMPLETED	2013-03-07 21:46:18
2	2	2	2	FAILED	2013-03-07 21:47:01
3	3	2	2	COMPLETED	2013-03-07 21:47:29

图 3-10 Job Execution 对应的数据表 BATCH_JOB_EXECUTION 的数据

Job Execution 对应的数据库表是：BATCH_JOB_EXECUTION，具体表的字段信息参见 3.5.4.3 节。

Job Execution 对应的 Java 类是 org.springframework.batch.core.JobExecution。

Job Execution 主要的属性描述信息参见表 3-2。

表 3-2 Job Execution 关键属性说明

属 性	说 明
status	BatchStatus 对象表示执行状态。BatchStatus.STARTED 表示运行时，BatchStatus.FAILED 表示执行失败，BatchStatus.COMPLETED 表示任务成功结束
startTime	表示任务开始时的系统时间；类型 java.util.Date
endTime	表示任务结束时的系统时间；类型 java.util.Date
exitStatus	ExitStatus 表示任务的运行结果，包含返回给调用者的退出代码
createTime	表示 Job Execution 第一次持久化时的系统时间。一个任务可能还没有启动（也就没有 startTime），但总是会有 createTime；类型 java.util.Date
lastUpdated	表示最近一次 Job Execution 被持久化的系统时间；类型 java.util.Date
executionContext	包含运行过程中所有需要被持久化的用户数据
failureException	在任务执行过程中例外的列表

3.3 Step

Step 表示作业中的一个完整步骤，一个 Job 可以由一个或者多个 Step 组成。Step 包含一个实际运行的批处理任务中的所有必需的信息，Step 可以是非常简单的业务实现，比如打印 HelloWorld 信息，也可以是非常复杂的业务处理，Step 的复杂程度通常是由业务决定的。

Step 与 Job 的关系参见图 3-11。

一个 Job 可以拥有一到多个 Step；一个 Step 可以有一到多个 Step Execution（当一个 Step 执行失败，下次重新执行该任务的时候，会为该 Step 重新生成一个 Step Execution）；一个 Job Execution 可以有一到多个 Step Execution（当一个 Job 由多个 Step 组成时，每个 Step 执行都会生成一个新的 Step Execution，则一个 Job Execution 会拥有多个 Step Execution）。

Step 中可以配置 tasklet、partition、job、flow，具体每个的作用后面章节会描述，这里仅罗列一下，给读者一个初步的印象。具体参见图 3-12。

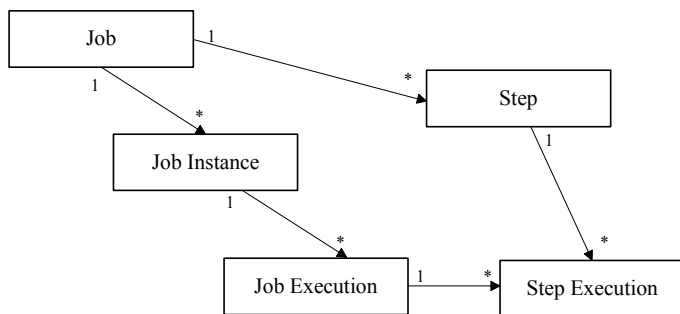


图 3-11 Job 与 Step 的关系

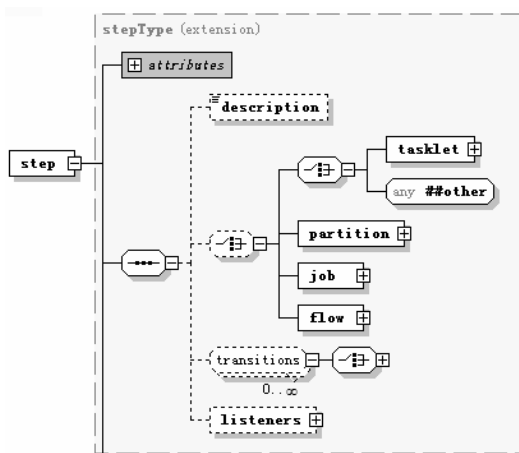


图 3-12 Step 元素的 Schema 定义

Step 的主要属性参见图 3-13。

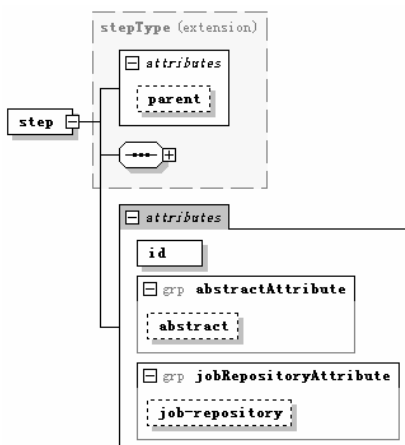


图 3-13 Step 元素的主要属性

具体的属性定义参见第 4 章的描述。

3.3.1 Step Execution

Step Execution 是 Step 执行的句柄。一次 Step 执行可能成功也可能失败。还记得 3.2.1 节我们执行的三次任务吗？我们的 billJob 只有一个 Step，但作业步执行器对应的表 BATCH_STEP_EXECUTION 却有三条记录。

表 BATCH_STEP_EXECUTION 中的记录参见图 3-14 和图 3-15。

	STEP_EXECUTION_ID	VERSION	STEP_NAME	JOB_EXECUTION_ID	STATUS
1	1	6	billStep	1	COMPLETED
2	2	3	billStep	2	FAILED
3	3	5	billStep	3	COMPLETED

图 3-14 Step Execution 对应的数据库表记录（一）

图 3-14 中的表数据如下。

START_TIME	END_TIME	COMMIT_COUNT	READ_COUNT	FILTER_COUNT
2013-03-07 21:46:18	2013-03-07 21:46:18	4	6	0
2013-03-07 21:47:01	2013-03-07 21:47:01	1	3	0
2013-03-07 21:47:29	2013-03-07 21:47:30	3	4	0

图 3-15 Step Execution 对应的数据库表记录（二）

具体描述如下：

第一条记录是执行参数为“20130308”的 billJob，唯一的 billStep 执行状态为完成；

第二条记录是执行参数为“20130309”的 billJob 且输入文件格式错误，导致本次的 billStep 执行状态为失败；

第三条记录是执行参数为“20130309”的 billJob 且输入文件格式正确，本次执行后，billStep 执行状态为成功。

Step Execution 对应的数据库表是：BATCH_STEP_EXECUTION。

具体表的字段信息参见 3.5.4.4 节。

Step Execution 对应的 Java 类是 org.springframework.batch.core.StepExecution。

Step 的执行过程是由 StepExecution 类的对象所表示的，包括每次执行所对应的 step、Job Execution、相关的事务操作（例如提交与回滚）、开始时间结束时间等。此外每次执行 step 时还包含一个 ExecutionContext，用来存放开发者在批处理运行过程中所需要的任何信息，例如用来重启的静态数据与状态数据。

表 3-3 列出了 StepExecution 的属性。

表 3-3 作业步执行器 StepExecution 属性说明

属 性	说 明
status	BatchStatus 对象表示了执行状态。BatchStatus.STARTED 表示运行时，BatchStatus.FAILED 表示执行失败，BatchStatus.COMPLETED 表示任务成功结束

续表

属 性	说 明
startTime	表示任务步开始时的系统时间；类型为 java.util.Date
endTime	表示任务步结束时的系统时间；类型为 java.util.Date
exitStatus	ExitStatus 表示任务步的运行结果，包含返回给调用者的退出代码
executionContext	在执行过程中任何需要进行持久化的用户数据
readCount	成功读取的记录数
writeCount	成功写入的记录数
commitCount	执行过程的事务中成功提交次数
rollbackCount	执行过程的事务中回滚次数
readSkipCount	读取失败而略过的记录数
processSkipCount	处理失败而略过的记录数
filterCount	被 ItemProcessor 过滤的记录数
writerSkipCount	写入失败而略过的记录数

3.4 Execution Context

Execution Context 是 Spring Batch 框架提供的持久化与控制的 key/value 对，能够让开发者在 Step Execution 或 Job Execution 中保存需要进行持久化的状态。我们在 3.2.1 节执行的三个作业，当以参数“20130309”且使用错误输入文件时，框架会每次 commit 后记录当前的提交记录数以及读的记录数，这样当 Step 发生错误时，下次重启 Job 会根据 Execution Context 中的数据恢复状态，保证继续从上次失败的点重新执行。

Step 的执行上下文数据存放在表 BATCH_STEP_EXECUTION_CONTEXT 中。

表 BATCH_STEP_EXECUTION_CONTEXT 中的记录参见图 3-16 和图 3-17。

	BATCH_STEP_EXECUTION_ID
1	1 [{"map":{"entry":{"string":"FlatFileItemWriter.current.count","long":366},
2	2 [{"map":{"entry":{"string":"FlatFileItemWriter.current.count","long":124},
3	3 [{"map":{"entry":{"string":"FlatFileItemWriter.current.count","long":366},

图 3-16 Execution Context 对应的数据库表记录（一）

SHORT_CONTEXT
{"string":"FlatFileItemWriter.written","long":6},{"string":"FlatFileItemReader.read.count","int":7}}}
{"string":"FlatFileItemWriter.written","long":2},{"string":"FlatFileItemReader.read.count","int":2}}}
{"string":"FlatFileItemWriter.written","long":4},{"string":"FlatFileItemReader.read.count","int":7}}}

图 3-17 Execution Context 对应的数据库表记录（二）

图 3-16 的数据如下。

第一条记录：成功写了 6 条记录；

第二条记录：成功写了 2 条记录；读者可以看一下我们输入错误文件 `ch03\data\credit-card-bill-201303-bad.csv` 是第四行有错误，同时我们在 `billJob` 中定义的提交间隔属性 `commit-interval="2"`，当第四行读错误时候，只有前两条写成功；

第三条记录：成功写了 4 条记录；修复输入文件第四行错误后，重新执行该 `Job`，会从上上次失败的点重新读取数据和写入数据（上次失败的地方是第 3 行，所以此次 `Job` 的执行从第三行开始），最终处理了 4 行数据。

`Execution Context` 分为两类：一类是 `Job Execution` 的上下文（对应表：`BATCH_JOB_EXECUTION_CONTEXT`）；另一类是 `Step Execution` 的上下文（对应表：`BATCH_STEP_EXECUTION_CONTEXT`）。两类上下文之间的关系：一个 `Job Execution` 对应一个 `Job Execution` 的上下文；每个 `Step Execution` 对应一个 `Step Execution` 上下文；同一个 `Job` 中的 `Step Execution` 共用 `Job Execution` 的上下文。因此如果同一个 `Job` 的不同 `Step` 间需要共享数据时，则可以通过 `Job Execution` 的上下文来共享数据。

3.5 Job Repository

Spring Batch 框架提供 `Job Repository` 来存储 `Job` 执行期的元数据（这里的元数据是指 `Job Instance`、`Job Execution`、`Job Parameters`、`Step Execution`、`Execution Context` 等数据），并提供两种默认实现。一种是存放在内存中（参见 2.3 节中 `Job Repository` 的定义）；另一种是将元数据存放在数据库中。通过将元数据存放在数据库中，可以随时监控批处理 `Job` 的执行状态，查看 `Job` 执行结果是成功还是失败，并且使得在 `Job` 失败的情况下重新启动 `Job` 成为可能。

Spring Batch 框架提供了可执行的数据库脚本和 `JobRepository` 的 CRUD 实现类。

（1）数据库脚本位置

`spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core`

（2）默认实现对元数据 CRUD 的操作类为：`org.springframework.batch.core.repository.support.SimpleJobRepository`。

本文中所有的例子使用 `MySQL` 数据库作为例子进行描述，读者根据需要可以使用其他的数据库。

说明：

Spring Batch 框架的 `JobRepository` 支持如下的数据库：`DB2`，`Derby`，`H2`，`HSQLDB`，`MySQL`，`Oracle`，`PostgreSQL`，`SQLServer`，`Sybase`。

3.5.1 Job Repository Schema

`Job Repository` 的 `Schema` 定义参见图 3-18。

`Job Repository Schema` 具体属性说明参见表 3-4。

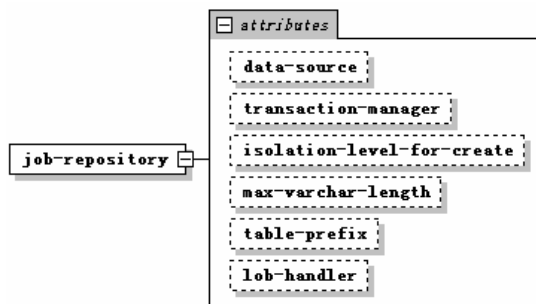


图 3-18 Job Repository 元素的 Schema 定义

表 3-4 作业仓库 Job Repository 属性说明

属 性	说 明
data-source	数据源 默认引用 dataSource
transaction-manager	事务管理器，如果使用了 namespace，repository 会被自动加上事务控制，这是为了确保批处理操作元数据以及失败后重启的状态能够被准确地持久化，如果 repository 的方法不是事务控制的，那么框架的行为就不能够被准确地定义。 默认引用 transactionManager
isolation-level-for-create	创建 job execution 实体时候隔离级别，默认使用 SERIALIZABLE，通常使用 REPEATABLE_READ 也可以很好地工作。 默认值：SERIALIZABLE
max-varchar-length	数据库字段类型为 VARCHAR 时候允许的最大长度。 默认值：2500
table-prefix	表名前缀，JobRepository 可以修改元数据表的表前缀，但是表名和列名不能修改。 默认值：BATCH_
lob-handler	大字段列的处理方式，默认只有 Oracle 数据库，或者 Spring Batch 框架不支持的数据库类型需要配置

3.5.2 配置 Memory Job Repository

Job Repository 来存储 Job 执行期的元数据，并提供两种默认实现。一种是存放在内存中，默认实现类为 MapJobRepositoryFactoryBean，另一种是存入在数据库中（见 3.5.3 节）。

Job Repository 内存配置在 HelloWorld2.3 节中已经给出，具体配置定义参见代码清单 3-4。

代码清单 3-4 Job Repository 内存配置定义

```

1.     <bean id="jobRepository"
2.         class="org.springframework.batch.core.repository.support.
3.             MapJobRepositoryFactoryBean">
  
```

通常在测试阶段可以使用基于内存的方式。

3.5.3 配置 DB Job Repository

Job Repository 用来存储 Job 执行期的元数据，另一种默认实现是存放在数据库中，通过将元数据存放在数据库中，可以随时监控批处理 Job 的执行状态，查看 Job 执行结果是成功还是失败，并且使得在 Job 失败的情况下重新启动 Job 成为可能。

说明：使用数据库的仓库时，需要首先根据 Spring Batch 框架提供的数据库脚本完成数据库的初始化，数据库脚本位置存放在 `spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core` 目录下，针对不同的数据库提供了不同的脚本，本书需要初始化文件：`schema-mysql.sql`。

Job Repository 数据库配置方式参见代码清单 3-5。

详细参见文件：`main/resources/ch03/job-context.xml`。

代码清单 3-5 Job Repository 数据库配置定义

```
1.      <job-repository id="jobRepository" data-source="dataSource"
2.          transaction-manager="transactionManager" isolation-level-for-
3.              create="SERIALIZABLE"
4.          table-prefix="BATCH_" max-varchar-length="1000"
5.      />
6.      <!-- 事务管理器 -->
7.      <bean:bean id="transactionManager"
8.          class="org.springframework.jdbc.datasource.DataSource
9.              TransactionManager">
10.
11.          <bean:property name="dataSource" ref="dataSource" />
12.      </bean:bean>
13.
14.      <!-- 数据源 -->
15.      <bean:bean id="dataSource"
16.          class="org.springframework.jdbc.datasource.DriverManager
17.              DataSource">
18.
19.          <bean:property name="driverClassName">
20.              <bean:value>com.mysql.jdbc.Driver</bean:value>
21.          </bean:property>
22.          <bean:property name="url">
23.              <bean:value>jdbc:mysql://127.0.0.1:3306/test</bean:value>
24.          </bean:property>
25.          <bean:property name="username" value="root"></bean:property>
26.          <bean:property name="password" value="000000"></bean:property>
27.      </bean:bean>
```

该程序说明如下。

1~4 行：定义 JobRepository，属性 data-source 定义使用的数据源为 dataSource，属性

transaction-manager 定义使用的事务管理器，属性 isolation-level-for-create 定义创建 Job Execution 时候的事务隔离级别，避免多个 Job Execution 执行一个 Job Instance，属性 table-prefix 定义使用的数据库表的前缀为 BATCH_，属性 max-varchar-length 定义 varchar 的最大长度为 1000。

5~9 行：定义数据管理器。

11~22 行：定义数据源，本例中定义 MySQL 类型的数据源。

3.5.4 数据库 Schema

Spring Batch 框架进行元数据管理共有九张表，其中有三张表（后缀为 SEQ）是用来分配主键的。九张表分别是：

BATCH_JOB_INSTANCE、
BATCH_JOB_EXECUTION、
BATCH_JOB_EXECUTION_PARAMS、
BATCH_STEP_EXECUTION、
BATCH_JOB_EXECUTION_CONTEXT、
BATCH_STEP_EXECUTION_CONTEXT、
BATCH_JOB_EXECUTION_SEQ、
BATCH_STEP_EXECUTION_SEQ、
BATCH_JOB_SEQ

读者可以从 Spring Batch 发布物找到具体的数据库脚本。例如，如果需要查看 MySQL 数据库的创建脚本可以参考：spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core\schema-mysql.sql。对应的删除数据库脚本在如下位置：spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core\schema-drop-mysql.sql。

在本书中，如果没有特别说明，使用的均是 MySQL 数据库。

表之间的关系参见图 3-19。

Spring Batch 具体的描述参见表 3-5。

表 3-5 Spring Batch 框架核心表说明

表 名	说 明
BATCH_JOB_INSTANCE	作业实例表，用于存放 Job 的实例信息
BATCH_JOB_EXECUTION_PARAMS	作业参数表，用于存放每个 Job 执行时候的参数信息，该参数实际上是对应 Job 实例的
BATCH_JOB_EXECUTION	作业执行器表，用于存放当前作业的执行信息，比如创建时间，执行开始时间，执行结束时间，执行的那个 Job 实例，执行状态等
BATCH_JOB_EXECUTION_CONTEXT	作业执行上下文表，用于存放作业执行器上下文的信息。

续表

表 名	说 明
BATCH_STEP_EXECUTION	作业步执行器表，用于存放每个 Step 执行器的信息，比如作业步开始执行时间，执行完成时间，执行状态，读/写次数，跳过次数等信息
BATCH_STEP_EXECUTION_CONTEXT	作业步执行上下文表，用于存放每个作业步上下文的信息
BATCH_JOB_SEQ	作业序列表，用于给表 BATCH_JOB_INSTANCE 和 BATCH_JOB_EXECUTION_PARAMS 提供主键
BATCH_JOB_EXECUTION_SEQ	作业执行器序列表，用于给表 BATCH_JOB_EXECUTION 和 BATCH_JOB_EXECUTION_CONTEXT 提供主键
BATCH_STEP_EXECUTION_SEQ	作业步序列表，用于给表 BATCH_STEP_EXECUTION 和 BATCH_STEP_EXECUTION_CONTEXT 提供主键

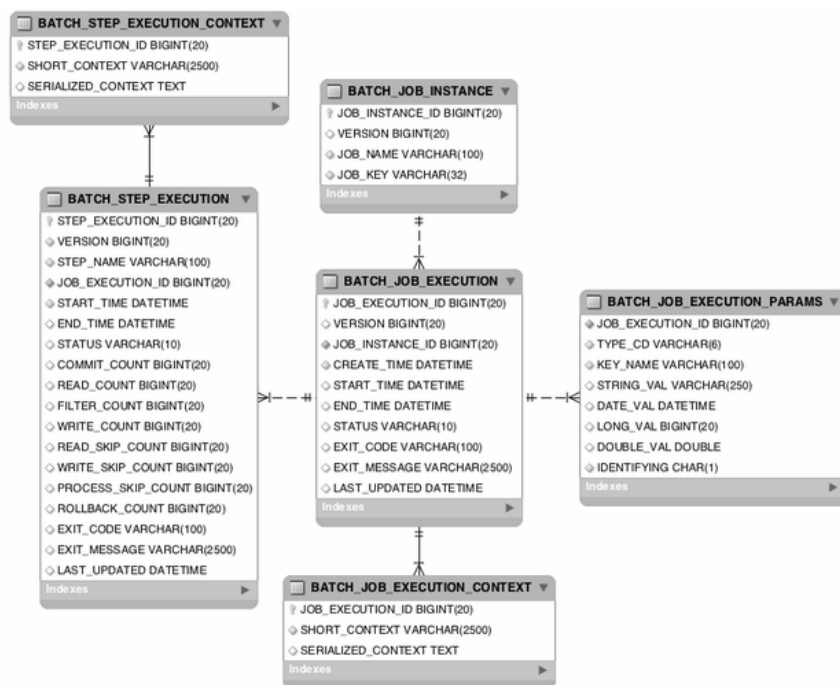


图 3-19 Spring Batch 框架核心数据库表关系图

接下来具体描述每张表的结构信息，以及字段的描述信息。

3.5.4.1 BATCH_JOB_INSTANCE

建表脚本参见代码清单 3-6。

代码清单 3-6 BATCH_JOB_INSTANCE 建表脚本

```
1. CREATE TABLE BATCH JOB INSTANCE (
2.     JOB_INSTANCE_ID BIGINT NOT NULL PRIMARY KEY ,
3.     VERSION BIGINT ,
4.     JOB_NAME VARCHAR(100) NOT NULL,
5.     JOB_KEY VARCHAR(32) NOT NULL,
6.     constraint JOB_INST_UN unique (JOB_NAME, JOB_KEY)
7. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_INSTANCE 字段信息描述参见表 3-6。

表 3-6 数据库表 BATCH_JOB_INSTANCE 字段说明

字 段	说 明
JOB_INSTANCE_ID	主键，作业实例 ID 编号，根据 BATCH_JOB_SEQ 自动生成
VERSION	版本号
JOB_NAME	作业名称，即在配置文件中定义的 Job id 字段内容
JOB_KEY	作业标识，根据作业参数序列化生成的标识；需要注意通过 JOB_NAME+ JOB_KEY 能够唯一区分一个作业实例。 如果是同一个 Job，则 JOB_KEY 一定不能相同，即作业参数不能相同；如果不是同一个 Job，JOB_KEY 则可以相同，即作业参数可以相同

3.5.4.2 BATCH_JOB_EXECUTION_PARAMS

建表脚本参见代码清单 3-7。

代码清单 3-7 BATCH_JOB_EXECUTION_PARAMS 建表脚本

```
1. CREATE TABLE BATCH_JOB_EXECUTION_PARAMS (
2.     JOB_EXECUTION_ID BIGINT NOT NULL ,
3.     TYPE_CD VARCHAR(6) NOT NULL ,
4.     KEY_NAME VARCHAR(100) NOT NULL ,
5.     STRING_VAL VARCHAR(250) ,
6.     DATE_VAL DATETIME DEFAULT NULL ,
7.     LONG_VAL BIGINT ,
8.     DOUBLE_VAL DOUBLE PRECISION ,
9.     IDENTIFYING CHAR(1) NOT NULL ,
10.    constraint JOB_EXEC_PARAMS_FK foreign key (JOB_EXECUTION_ID)
11.    references BATCH_JOB_EXECUTION (JOB_EXECUTION_ID)
12. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_EXECUTION_PARAMS 字段信息描述参见表 3-7。

表 3-7 数据库表 BATCH_JOB_EXECUTION_PARAMS 字段说明

字 段	说 明
JOB_EXECUTION_ID	外键，作业执行器 ID 编号，一个作业实例可能会有多行参数记录，主要根据参数的个数决定

续表

字 段	说 明
TYPE_CD	参数的类型，可能是如下四种当中的一种：date、string、long、double
KEY_NAME	参数名字
STRING_VAL	参数如果是 string，此列存放 string 类型参数值
DATE_VAL	参数如果是 date，此列存放 date 类型参数值
LONG_VAL	参数如果是 long，此列存放 long 类型参数值
DOUBLE_VAL	参数如果是 double，此列存放 double 类型参数值
IDENTIFYING	用于标识作业参数是否标识作业实例

3.5.4.3 BATCH_JOB_EXECUTION

建表脚本参见代码清单 3-8。

代码清单 3-8 BATCH_JOB_EXECUTION 建表脚本

```

1. CREATE TABLE BATCH_JOB_EXECUTION (
2.     JOB_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY ,
3.     VERSION BIGINT ,
4.     JOB_INSTANCE_ID BIGINT NOT NULL,
5.     CREATE_TIME DATETIME NOT NULL,
6.     START_TIME DATETIME DEFAULT NULL ,
7.     END_TIME DATETIME DEFAULT NULL ,
8.     STATUS VARCHAR(10) ,
9.     EXIT_CODE VARCHAR(100) ,
10.    EXIT_MESSAGE VARCHAR(2500) ,
11.    LAST_UPDATED DATETIME,
12.    constraint JOB_INST_EXEC_FK foreign key (JOB_INSTANCE_ID)
13.    references BATCH_JOB_INSTANCE (JOB_INSTANCE_ID)
14. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_EXECUTION 字段信息描述参见表 3-8。

表 3-8 数据库表 BATCH_JOB_EXECUTION 字段说明

字 段	说 明
JOB_EXECUTION_ID	主键，作业执行器 ID 编号
VERSION	版本号
JOB_INSTANCE_ID	作业实例 ID 编号
CREATE_TIME	作业执行器创建时间
START_TIME	作业执行器开始执行时间
END_TIME	作业执行器结束时间

续表

字 段	说 明
STATUS	作业执行器执行的状态，如：COMPLETED, STARTING, STARTED, STOPPING, STOPPED, FAILED, ABANDONED, UNKNOWN。 这些状态在类 org.springframework.batch.core.BatchStatus 中定义
EXIT_CODE	作业执行器退出编码，如：UNKNOWN, EXECUTING, COMPLETED, NOOP, FAILED, STOPPED。 这些状态在类 org.springframework.batch.core.ExitStatus 中定义
EXIT_MESSAGE	作业执行器退出描述，详细描述退出的信息，如果发生了异常，通常包含异常的堆栈信息
LAST_UPDATED	本条记录上次更新时间

3.5.4.4 BATCH_STEP_EXECUTION

建表脚本参见代码清单 3-9。

代码清单 3-9 BATCH_STEP_EXECUTION 建表脚本

```

1. CREATE TABLE BATCH_STEP_EXECUTION (
2.     STEP_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY ,
3.     VERSION BIGINT NOT NULL,
4.     STEP_NAME VARCHAR(100) NOT NULL,
5.     JOB_EXECUTION_ID BIGINT NOT NULL,
6.     START_TIME DATETIME NOT NULL ,
7.     END_TIME DATETIME DEFAULT NULL ,
8.     STATUS VARCHAR(10) ,
9.     COMMIT_COUNT BIGINT ,
10.    READ_COUNT BIGINT ,
11.    FILTER_COUNT BIGINT ,
12.    WRITE_COUNT BIGINT ,
13.    READ_SKIP_COUNT BIGINT ,
14.    WRITE_SKIP_COUNT BIGINT ,
15.    PROCESS_SKIP_COUNT BIGINT ,
16.    ROLLBACK_COUNT BIGINT ,
17.    EXIT_CODE VARCHAR(100) ,
18.    EXIT_MESSAGE VARCHAR(2500) ,
19.    LAST_UPDATED DATETIME,
20.    constraint JOB_EXEC_STEP_FK foreign key (JOB_EXECUTION_ID)
21.    references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
22. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_EXECUTION 字段信息描述参见表 3-9。

表 3-9 数据库表 BATCH_STEP_EXECUTION 字段说明

字 段	说 明
STEP_EXECUTION_ID	主键，作业步实例 ID 编号
VERSION	版本号
STEP_NAME	作业步名字
JOB_EXECUTION_ID	外键，作业执行器 ID
START_TIME	作业步执行器开始执行时间
END_TIME	作业步执行器结束时间
STATUS	作业步执行器执行的状态，如：COMPLETED，STARTING，STARTED，STOPPING，STOPPED，FAILED，ABANDONED，UNKNOWN。 这些状态在类 org.springframework.batch.core.BatchStatus 中定义
COMMIT_COUNT	事务提交次数
READ_COUNT	读数据的次数
FILTER_COUNT	过滤掉的数据次数
WRITE_COUNT	写数据的次数
READ_SKIP_COUNT	读数据跳过的次数
WRITE_SKIP_COUNT	写数据跳过的次数
PROCESS_SKIP_COUNT	处理数据跳过的次数
ROLLBACK_COUNT	事务回滚次数
EXIT_CODE	作业步执行器退出编码，如：UNKNOWN，EXECUTING，COMPLETED，NOOP，FAILED，STOPPED。 这些状态在类 org.springframework.batch.core.ExitStatus 中定义
EXIT_MESSAGE	作业步执行器退出描述，详细描述退出的信息，如果发生了异常，通常包含异常的堆栈信息
LAST_UPDATED	本条记录上次更新时间

3.5.4.5 BATCH_JOB_EXECUTION_CONTEXT

建表脚本参见代码清单 3-10。

代码清单 3-10 BATCH_JOB_EXECUTION_CONTEXT 建表脚本

```

1. CREATE TABLE BATCH_JOB_EXECUTION_CONTEXT (
2.     JOB_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
3.     SHORT_CONTEXT VARCHAR(2500) NOT NULL,
4.     SERIALIZED_CONTEXT TEXT ,
5.     constraint JOB_EXEC_CTX_FK foreign key (JOB_EXECUTION_ID)
6.     references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
7. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_EXECUTION_CONTEXT 字段信息描述参见表 3-10。

表 3-10 数据库表 BATCH_JOB_EXECUTION_CONTEXT 字段说明

字 段	说 明
JOB_EXECUTION_ID	外键，作业执行器 ID 编号
SHORT_CONTEXT	作业执行器上下文字符串格式
SERIALIZED_CONTEXT	序列化的作业执行器上下文

3.5.4.6 BATCH_STEP_EXECUTION_CONTEXT

建表脚本参见代码清单 3-11。

代码清单 3-11 BATCH_STEP_EXECUTION_CONTEXT 建表脚本

```

1. CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT (
2.     STEP_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
3.     SHORT_CONTEXT VARCHAR(2500) NOT NULL,
4.     SERIALIZED_CONTEXT TEXT ,
5.     constraint STEP_EXEC_CTX_FK foreign key (STEP_EXECUTION_ID)
6.     references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
7. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_EXECUTION_CONTEXT 字段信息描述参见表 3-11。

表 3-11 数据库表 BATCH_STEP_EXECUTION_CONTEXT 字段说明

字 段	说 明
STEP_EXECUTION_ID	外键，作业步执行器 ID 编号
SHORT_CONTEXT	作业执行器上下文字符串格式
SERIALIZED_CONTEXT	序列化的作业执行器上下文

3.6 Job Launcher

Job Launcher（作业调度器）是 Spring Batch 框架基础设施层提供的运行 Job 的能力。通过给定的 Job 名称和作业参数 Job Parameters，可以通过 Job Launcher 执行 Job。通过 Job Launcher 可以在 Java 程序中调用批处理任务，也可以在通过命令行或者其他框架（如定时调度框架 Quartz）中调用批处理任务。Spring Batch 框架提供了 Job Launcher 的简单实现 SimpleJobLauncher。

JobLauncher 接口定义参见代码清单 3-12。

代码清单 3-12 JobLauncher 接口定义

```

1. public interface JobLauncher {
2.     public JobExecution run(Job job, JobParameters jobParameters) throws
3.         JobExecutionAlreadyRunningException,
4.         JobRestartException,
5.         JobInstanceAlreadyCompleteException,
```

```

6.             JobParametersInvalidException;
7.    }

```

该接口只有一个 `run` 方法，两个参数 `job` 和 `job Parameters`。批处理应用可以通过 `Job Launcher` 和外部系统交互，通常情况下外部系统可以同步也可以异步调用批处理应用；批处理应用本身可以访问外部的资源，例如数据库、文件、消息队列等。

3.7 ItemReader

`ItemReader` 是 `Step` 中对资源的读处理，`Spring Batch` 框架已经提供了多种类型的读实现，包括对文本文件、XML 文件、数据库、JMS 消息等读的处理。直接使用 `Spring Batch` 框架提供的读组件可以快速地完成批处理应用的开发和搭建。同时 `Spring Batch` 框架提供了较好的复用和扩展机制，可以复用现有的读服务或者快速实现自定义的读组件。

`Spring Batch` 框架提供的读组件列表参见表 3-12。

表 3-12 `Spring Batch` 框架提供的 `ItemReader` 组件

ItemReader	说 明
<code>ListItemReader</code>	读取 <code>List</code> 类型数据，只能读一次
<code>ItemReaderAdapter</code>	<code>ItemReader</code> 适配器，可以复用现有的读操作
<code>FlatFileItemReader</code>	读 <code>Flat</code> 类型文件
<code>StaxEventItemReader</code>	读 XML 类型文件
<code>JdbcCursorItemReader</code>	基于 <code>JDBC</code> 游标方式读数据库
<code>HibernateCursorItemReader</code>	基于 <code>Hibernate</code> 游标方式读数据库
<code>StoredProcedureItemReader</code>	基于存储过程读数据库
<code>IbatisPagingItemReader</code>	基于 <code>Ibatis</code> 分页读数据库
<code>JpaPagingItemReader</code>	基于 <code>Jpa</code> 方式分页读数据库
<code>JdbcPagingItemReader</code>	基于 <code>JDBC</code> 方式分页读数据库
<code>HibernatePagingItemReader</code>	基于 <code>Hibernate</code> 方式分页读取数据库
<code>JmsItemReader</code>	读取 <code>JMS</code> 队列
<code>IteratorItemReader</code>	迭代方式的读组件
<code>MultiResourceItemReader</code>	多文件读组件
<code>MongoItemReader</code>	基于分布式文件存储的数据库 <code>MongoDB</code> 读组件
<code>Neo4jItemReader</code>	面向网络的数据库 <code>Neo4j</code> 读组件
<code>ResourcesItemReader</code>	基于批量资源的读组件，每次读取返回资源对象
<code>AmqpItemReader</code>	读取 <code>AMQP</code> 队列组件
<code>RepositoryItemReader</code>	基于 <code>Spring Data</code> 的读组件

上面所有的组件均实现 `ItemReader` 接口，接口定义参见代码清单 3-13。

代码清单 3-13 ItemReader 接口定义

```
1. public interface ItemReader<T> {
2.     T read() throws Exception,
3.         UnexpectedInputException,
4.         ParseException,
5.         NonTransientResourceException;
6. }
```

read 方法负责从给定的资源中读取数据。

3.8 ItemProcessor

ItemProcessor 阶段表示对读取的数据进行处理，开发者可以实现自己的业务操作来对数据进行处理。

Spring Batch 框架提供的处理组件列表参见表 3-13。

表 3-13 Spring Batch 框架提供的 ItemProcessor 组件

ItemProcessor	说 明
CompositemItemProcessor	组合处理器，可以封装多个业务处理服务
ItemProcessorAdapter	ItemProcessor 适配器，可以复用现有的业务处理服务
PassThroughItemProcessor	不做任何业务处理，直接返回读到的数据
ValidatingItemProcessor	数据校验处理器，支持对数据的校验，如果校验不通过可以进行过滤掉或者通过 skip 的方式跳过对记录的处理

业务操作需要实现 ItemProcessor 接口，接口定义参见代码清单 3-14 ItemProcessor 接口定义。

代码清单 3-14 ItemProcessor 接口定义

```
1. public interface ItemProcessor<I, O> {
2.     O process(I item) throws Exception;
3. }
```

process 方法中，参数 item 是 ItemReader 读取的数据，返回值 O 是交给 ItemWriter 写的的数据，在 process 方法中可以修改读到的数据的值。如果返回值是 null，表示忽略这次的数据，具体描述参见 Item Processor 对应章节的描述。

3.9 ItemWriter

ItemWriter 是 Step 中对资源的写处理，Spring Batch 框架已经提供了多种类型的写实现，包括对文本文件、XML 文件、DB 等写的处理。直接使用 Spring Batch 框架提供的写组件可以快速地完成应用的开发和搭建。同时 Spring Batch 框架提供了较好的复用和扩展机制，可以

复用现有的写服务或者快速实现自定义的写组件。

Spring Batch 框架提供的写组件列表参见表 3-14。

表 3-14 Spring Batch 框架提供的 ItemWriter 组件

ItemWriter	说 明
FlatFileItemWriter	写 Flat 类型文件
MultiResourceItemWriter	多文件写组件
StaxEventItemWriter	写 XML 类型文件
AmqpItemWriter	写 AMQP 类型消息
ClassifierCompositItemWriter	根据 Classifier 路由不同的 Item 到特定的 ItemWriter 处理
HibernateItemWriter	基于 Hibernate 方式写数据库
IbatisBatchItemWriter	基于 Ibatis 方式写数据库
ItemWriterAdapter	ItemWriter 适配器，可以复用现有的写服务
JdbcBatchItemWriter	基于 JDBC 方式写数据库
JmsItemWriter	写 JMS 队列
JpaItemWriter	基于 Jpa 方式写数据库
GemfireItemWriter	基于分布式数据库 Gemfire 的写组件
SpELMappingGemfireItemWriter	基于 Spring 表达式语言写分布式数据库 Gemfire 的组件
MimeMessageItemWriter	发送邮件的写组件
MongoItemWriter	基于分布式文件存储的数据库 MongoDB 写组件
Neo4jItemWriter	面向网络的数据库 Neo4j 的读组件
PropertyExtractingDelegatingItemWriter	属性抽取代理写组件；通过调用给定的 Spring Bean 方法执行写入，参数由 Item 中指定的属性字段获取作为参数
RepositoryItemWriter	基于 Spring Data 的写组件
SimpleMailMessageItemWriter	发送邮件的写组件
CompositItemWriter	条目写的组合模式，支持组装多个 ItemWriter

上面的组件均实现了 ItemWriter 接口，接口定义参见代码清单 3-15。

代码清单 3-15 ItemWriter 接口定义

```
1. public interface ItemWriter<T> {
2.     void write(List<? extends T> items) throws Exception;
3. }
```

write 方法负责将数据写入到给定的资源中。

第 4 章我们将详细描述如何配置 Job 和运行 Job。

配置作业 Job

本章详细描述如何配置 Job 及如何运行 Job。在配置 Job 部分，我们将配置重启特性、抽象特性、作业仓库、监听器、参数校验等功能。在运行 Job 部分我们将描述如何运行已经配置好的 Job。

第 3 章中我们描述了批处理的整体架构，在进入具体配置 Job 前，我们重新复习一下批处理的整体架构，参见图 4-1。一个 Job 由 1 个或者多个 Step 组成，Step 有读、处理、写三部分操作组成；Job 运行期所有的数据通过 Job Repository 进行持久化，同时通过 JobLauncher 负责调度 Job 作业。

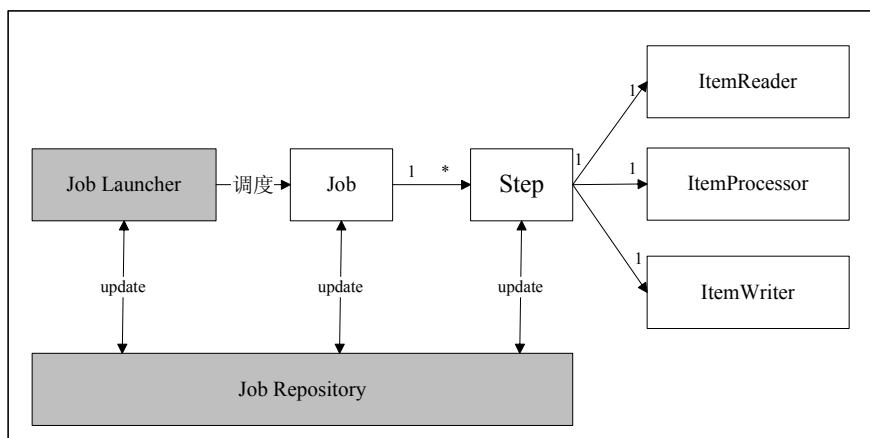


图 4-1 Spring Batch 批处理框架架构图

为了简化 Spring Batch 的配置，Spring Batch 框架提供了独立的 XML 配置，开发者无需再以 Spring Bean 的方式进行配置，提供了独立的 job、step、tasklet、chunk 标签。图 4-2 展示了 Spring Batch 框架提供的 XML 元素和属性，基于这些元素，开发者可以快速、高效地进行批处理应用的开发。

4.1 基本配置

在配置 Job 之前，我们一起看一下 Job 的主要属性定义和元素定义。

Job 主要属性包括 id（作业唯一标识）、job-repository（定义作业仓库）、incrementer（作业参数递增器）、restartable（作业是否可以重启）、parent（指定该作业的父作业）、abstract

(定义作业是否抽象的)。图 4-3 展示了 Job 属性的 Schema 的定义，表 4-1 给出了 Job 属性的详细说明。

element	job
element	step
element	flow
element	job-listener
element	step-listener
element	job-repository
group	flowGroup
complexType	stepType
complexType	partitionType
complexType	taskletType
complexType	transaction-attributesType
simpleType	isolationType
group	beanElementGroup
complexType	chunkTaskletType
attributeGroup	nextAttribute
attributeGroup	exceptionClassAttribute
group	includeElementGroup
group	includeExcludeElementGroup
complexType	listenerType
complexType	jobExecutionListenerType
complexType	stepListenerType
complexType	stepListenersType
group	transitions
attributeGroup	jobRepositoryAttribute
attributeGroup	parentAttribute
attributeGroup	abstractAttribute
attributeGroup	mergeAttribute
attributeGroup	adapterMethodAttribute
simpleType	description

图 4-2 Spring Batch 框架提供的 XML 元素和属性

Job 主要子元素包括 step (定义作业步)、split (定义并行作业步)、flow (引用独立定义的作业流)、decision (定义作业步执行的条件判断器, 用于判断后续执行的作业步)、listeners (定义作业拦截器)、validator (定义作业参数校验器)。图 4-4 展示了 Job 子元素的 Schema 的定义，表 4-2 给出了 Job 子元素的详细说明。

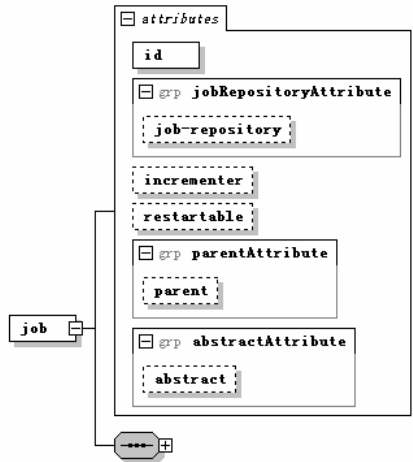


图 4-3 Job 属性 Schema 的定义

表 4-1 Job 属性说明

属 性	说 明	默 认 值
id	Job 的唯一标识，在整个运行上下文中不允许重复	
job-repository	定义该 Job 运行期间使用的 Job 仓库，默认使用名字为 jobRepository 的 Bean	jobRepository
incrementer	作业参数递增器，只有在 org.springframework.batch.core.relaunch.JobOperator 的 startNextInstance 方法中使用	
restartable	定义当前作业是否支持重启，默认值是 true，表示支持重启，如果不需要重启，需要显示设置为 false	true
parent	定义当前 Job 的父 Job。Job 可以从其他 Job 继承。通常在父 Job 中定义共有的属性；在子 Job 中定义特有的属性	
abstract	定义当前 Job 是否是抽象的。True 表示当前 Job 是抽象的，不能被实例化	

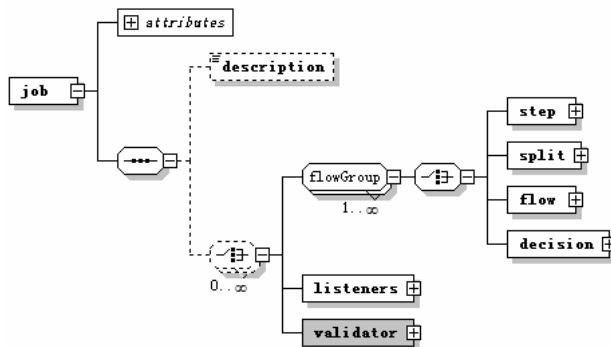


图 4-4 Job 子元素 Schema 的定义

表 4-2 Job 子元素说明

属 性	说 明
step	定义 Job 的作业步
split	定义并行的 Step
flow	引用独立配置的作业步流程
decision	Step 执行的条件判断器，根据 decision 可以动态地决定后续执行的 Step
listeners	定义 Job 执行时的拦截器
validator	定义 JobParameters 的验证器

4.1.1 重启 Job

通过属性 `restartable` 可以定义 Job 是否可以重启。默认情况下 Job 是可以重启的，需要注意的是，即使配置了 Job 可以重新启动，仍需要保证 Job Instance 的状态一定不能为“COMPLETED”状态。代码清单 4-1 展示了如何配置 Job 不能重启：

代码清单 4-1 配置 Job 不能重启

```
1.     <job id="billJob" restartable="false">
2.         <step id="billStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="csvItemReader" writer="csvItemWriter"
5.                     processor="creditBillProcessor" commit-interval="2">
6.                     </chunk>
7.                 </tasklet>
8.             </step>
9.         </job>
```

该程序说明如下。

1 行：属性 `restartable` 的默认值为 `true`，表示支持重新启动；如果不需要支持重启，需要显示设该值为 `false`。

执行示例项目中的类 `test.com.juxtapose.example.ch04.JobLaunchRestart` 两次，第一次执行因为对应的输入文件格式错误，对应的 `Job Instance` 的状态是“`FAILED`”；第二次重新执行类 `test.com.juxtapose.example.ch04.JobLaunchRestart`，虽然对应的 `Job Instance` 的状态是“`FAILED`”，正常情况下可以重启 `Job`，但因为该 `Job` 配置为不可以重启，所以重新启动该 `Job` 会抛出异常 `org.springframework.batch.core.repository.JobRestartException`。

重新启动不支持重启的 `Job`，会得到代码清单 4-2 中的异常信息。

代码清单 4-2 重新启动无法重启 Job 异常信息

```
1. org.springframework.batch.core.repository.JobRestartException:
2.     JobInstance already exists and is not restartable
3.     at org.springframework.batch.core.launch.support.SimpleJobLauncher.
4.         run(SimpleJobLauncher.java:97)
5.     at test.com.juxtapose.example.ch04.JobLaunchRestart.executeJob
6.         (JobLaunch.java:31)
7.     at test.com.juxtapose.example.ch04.JobLaunchRestart.main(JobLaunch.
8.         java:42)
```

4.1.2 Job 拦截器

Spring Batch 框架在 `Job` 执行阶段提供了拦截器，使得在 `Job` 执行前后能够加入自定义的业务逻辑处理。`Job` 执行阶段拦截器需要实现接口：`org.springframework.batch.core.JobExecutionListener`。`JobExecutionListener` 接口参见代码清单 4-3。

代码清单 4-3 `JobExecutionListener` 接口定义

```
1. public interface JobExecutionListener {
2.     void beforeJob(JobExecution jobExecution);
3.     void afterJob(JobExecution jobExecution);
4. }
```


该程序说明如下。

2 行：表示在 Job 执行之前调用该方法。

3 行：表示在 Job 执行之后调用该方法。

通过属性 listeners 可以为 Job 指定拦截器列表，拦截器可以一个或者多个，如有多个，则拦截器执行顺序为配置文件中定义的顺序。代码清单 4-4 给出了作业 billJob 的拦截器配置。

代码清单 4-4 Job 拦截器配置

```
1.     <job id="billJob">
2.         <step id="billStep">
3.             .....
4.         </step>
5.         <listeners>
6.             <listener ref="sysoutListener"></listener>
7.         </listeners>
8.     </job>
9.     <bean:bean id="sysoutListener"
10.         class="com.juxtapose.example.ch04.listener.SystemOutJobExecution
11.             Listener">
12.     </bean:bean>
```

billJob 执行时，listeners 中定义的拦截器 SystemOutJobExecutionListener 会被执行，该拦截器仅做了打印输出。读者可以根据具体的业务需要实现自定义的作业拦截器。SystemOutJobExecutionListener 的实现参见示例项目代码 com.juxtapose.example.ch04.listener.SystemOutJobExecutionListener。

Spring Batch 框架默认提供 JobExecutionListener 的实现，CompositeJobExecutionListener（拦截器组合器，支持配置一组拦截器）、JobExecutionListenerSupport（拦截器空实现，让开发者快速实现关心的业务操作）分别完成不同的功能，表 4-3 给出了框架的默认实现。

表 4-3 JobExecutionListener 默认实现。

JobExecutionListener 默认实现	功能说明
CompositeJobExecutionListener	拦截器组合模式，支持一组拦截器调用
JobExecutionListenerSupport	JobExecutionListener 空实现，可以直接继承，仅覆写关心的方法

拦截器异常

拦截器方法如果抛出异常会影响 Job 的正常执行，所以在执行自定义的拦截器时候，需要考虑对拦截器发生的异常做处理，避免影响业务。

如果拦截器发生异常，会导致 Job 执行的状态为"FAILED"。

执行顺序

在配置文件中可以配置多个 listener，拦截器之间的执行顺序按照 listener 定义的顺序执

行。before 方法按照 listener 定义的顺序执行，after 方法按照相反的顺序执行。代码清单 4-5 中执行顺序如下：

1. sysoutListener 拦截器的 before 方法；
2. sysoutAnnotationListener 拦截器的 before 方法；
3. sysoutAnnotationListener 拦截器的 after 方法；
4. sysoutListener 拦截器的 after 方法。

代码清单 4-5 Job 多拦截器配置

```
1. <job id="billJob" restartable="false">
2.     <step id="billStep">
3.         .....
4.     </step>
5.     <listeners>
6.         <listener ref="sysoutListener"></listener>
7.         <listener ref="sysoutAnnotationListener"></listener>
8.     </listeners>
9. </job>
```

Annotation

Spring Batch 框架提供了 Annotation 机制，可以不实现接口 JobExecutionListener，直接通过 Annotation 的机制定义拦截器。为 JobExecutionListener 提供的 Annotation 有：

- @BeforeJob
- @AfterJob

使用 Annotation 声明的拦截器和实现接口 JobExecutionListener 的拦截器，对应的配置格式一致，只需要在 listeners 节点中声明即可。

代码清单 4-6 展示了通过 Annotation 声明的拦截器代码。

代码清单 4-6 通过 Annotation 定义 Job 拦截器

```
1. public class SystemOut {
2.     @BeforeJob
3.     public void beforeJob(JobExecution jobExecution) {
4.         System.out.println("Annotation: JobExecution creat time:" +
5.                             jobExecution.getCreateTime());
6.         //throw new RuntimeException("listener make error!");
7.     }
8.     @AfterJob
9.     public void afterJob(JobExecution jobExecution) {
10.        System.out.println("Annotation: Job execute state:" +
11.                             jobExecution.getStatus().toString());
12.    }
13. }
```

该程序的说明如下。

2 行: `@BeforeJob` 声明作业执行前的操作。

8 行: `@AfterJob` 声明作业执行后的操作。

4.1.3 Job Parameters 校验

Spring Batch 框架提供了 Job 作业参数的校验功能，在 3.2.2 节中给出了 Job Parameters 详细介绍，读者如有遗忘，可以重新阅读该节的内容。开发者可以自定义实现参数校验器（需要实现接口：`org.springframework.batch.core.JobParametersValidator`）或者使用框架已经提供的校验组件类。通过参数校验，可以保证 Job 执行可以按照给定的参数执行，避免出现不必要的逻辑错误。

Spring Batch 框架默认提供了 `JobParametersValidator` 的实现，`CompositeJobParametersValidator`（参数校验组合器，支持配置一组参数校验类）、`DefaultJobParametersValidator`（参数校验默认实现类，支持必填校验和可选校验）分别完成不同功能，表 4-4 给出了 Spring Batch 框架提供的参数校验组件。

表 4-4 `JobParametersValidator` 默认实现

JobParametersValidator 默认实现	功能说明
<code>CompositeJobParametersValidator</code>	参数校验组合模式，支持一组参数校验
<code>DefaultJobParametersValidator</code>	参数校验默认实现，支持必须输入的参数和可以选择输入的参数

通过属性 `validator` 定义参数校验实现类，代码清单 4-7 展示了如何为参数配置参数校验。

代码清单 4-7 Job Parameters 配置

```
1.     <job id="billJob">
2.         <step id="billStep">
3.             .....
4.         </step>
5.         <listeners>
6.             <listener ref="sysoutListener"></listener>
7.             <listener ref="sysoutAnnotationListener"></listener>
8.         </listeners>
9.         <validator ref="validator"></validator>
10.    </job>
11.    <!-- 参数校验 -->
12.    <bean:bean id="validator"
13.        class="org.springframework.batch.core.job.DefaultJob
14.            ParametersValidator" >
15.        <bean:property name="requiredKeys" >
```

```

15.         <bean:set>
16.             <bean:value>date</bean:value>
17.         </bean:set>
18.     </bean:property>
19.     <bean:property name="optionalKeys" >
20.         <bean:set>
21.             <bean:value>name</bean:value>
22.         </bean:set>
23.     </bean:property>
24. </bean:bean>

```

该程序的说明如下。

9 行：通过属性 `validator` 定义参数校验使用的类对象。

14~18 行：通过关键字 `requiredKeys` 定义必须输入的参数 `date`。

19~23 行：通过关键字 `optionalKeys` 定义可以选择输入的参数 `name`。

因此执行该 Job 时候，必须输入 `date` 参数，最多输入 `date`、`name` 两个参数。任何其他参数名字都会导致参数校验类不通过。

在执行 Job 时候，可以通过 `JobParametersBuilder` 构造作业参数。代码清单 4-8 展示了输入不同参数执行 `billJob` 的示例。

代码清单 4-8 输入不同参数执行 Job

```

1. executeJob("ch04/job/job-validator.xml", "billJob",
2.     new JobParametersBuilder().addDate("date", new Date()));
3.
4. executeJob("ch04/job/job-validator.xml", "billJob",
5.     new JobParametersBuilder().addDate("date", new Date())
6.     .addString("test", "test parameter not allowed."));

```

该程序的说明如下。

1~2 行：`billJob` 可以正确执行，因为只输入了参数“date”

4~6 行：执行 `billJob` 会发生错误，会抛出异常 `org.springframework.batch.core.JobParametersInvalidException`，这是因为输入了非法的参数“test”。

4.1.4 Job 抽象与继承

Spring Batch 框架支持抽象的 Job 定义和 Job 的继承特性。通过定义抽象的 Job 可以将 Job 的共性进行抽取，形成父类的 Job 定义，父 Job 通常具有较多的共性；然后各个具体的 Job 可以继承父类 Job 的特性，并定义自己的属性。通过 Job 的属性 `abstract` 可以定义抽象的 Job，通过属性 `parent` 可以指定当前 Job 的父 Job。

Job 的抽象、继承与 Java 中的抽象、继承概念比较类似。父不能被实例化、子可以继承父的所有特性，甚至可以覆写掉父中的特性。

抽象 Job

通过 `abstract` 属性可以指定 Job 为抽象的 Job。抽象的 Job 不能被实例化，只能作为其他 Job 的父。通常可以在抽象 Job 中定义全局性的配置，供子 Job 使用。代码清单 4-9 给出了抽象 Job 配置。

代码清单 4-9 抽象 Job 配置

```
1.     <job id="baseJob" abstract="true">
2.         <listeners>
3.             <listener ref="sysoutListener"></listener>
4.         </listeners>
5.     </job>
```

其中，1 行：通过 `abstract` 属性定义 `baseJob` 为抽象的 Job，注意抽象的 Job 不能被实例化，任何试图直接调用 `baseJob` 的用例都会发生错误。

继承 Job

通过 `parent` 属性可以指定当前 Job 的父类，类似于 Java 世界中的继承一样，子类 Job 具有父类中定义的所有属性能力。子类继承时候，可以从抽象 Job 继承，也可以从普通的 Job 中继承。图 4-5 展示了 Job 间的继承关系，`billJob` 继承了抽象的 `baseJob`，多个 Job 可以继承自同一个 Job。

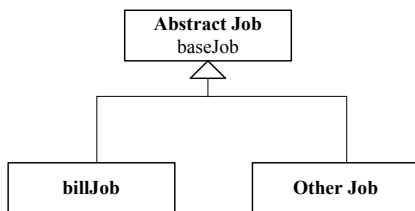


图 4-5 Job 继承

假设有这样一个场景，所有的 Job 都希望拦截器 `sysoutListener` 能够执行，而拦截器 `sysoutAnnotationListener` 则由每个具体的 Job 定义是否执行，通过抽象和继承属性可以完成上面的场景。

在抽象 `baseJob` 中定义 `sysoutListener`，在具体的 `billJob` 中定义 `sysAnnotationListener`，代码清单 4-10 给出了 `billJob` 的配置。

代码清单 4-10 继承 Job 配置

```
1.     <job id="billJob" parent="baseJob">
2.         <step id="billStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="csvItemReader" writer="csvItemWriter"
5.                     processor="creditBillProcessor" commit-interval="2">
6.                 </chunk>
```

```

7.         </tasklet>
8.     </step>
9.     <listeners merge="true">
10.         <listener ref="sysoutAnnotationListener"></listener>
11.     </listeners>
12. </job>

```

该程序说明如下。

1 行：通过属性 `parent` 指定当前 `billJob` 的父，通过集成 `baseJob`，`billJob` 拥有父类 `baseJob` 的所有特性，因此在 `billJob` 执行的时候也会调用 `baseJob` 中定义的拦截器 `sysoutListener`。

9 行：通过 `merge` 属性，可以与父类中的拦截器配置进行合并，表示在 `billJob` 中有两个拦截器会同时工作

merge 列表

如果父子中均定义了拦截器，则可以通过设置 `merge` 属性为 `true` 对拦截器列表合并；如果设置 `merge` 属性为 `false`，则子类中定义的拦截器直接覆盖掉父类中定义的拦截器。

通过抽象和继承的特性可以方便地对 `Job` 进行共性抽取和 `Job` 的复用。

4.2 高级特性

4.2.1 Step Scope

什么是 `Scope`（作用域、范围、生命周期）？

`Scope` 用来声明 `IOC` 容器中对象的存活空间，即在 `IOC` 容器在对象进入相应的 `Scope` 之前，生成并装配这些对象，在该对象不再处于这些 `Scope` 的限定范围之后，容器通常会销毁这些对象。

`Step Scope` 是 `Spring Batch` 框架提供了自定义的 `Scope`，将 `Spring Bean` 定义为 `Step Scope`，支持 `Spring Bean` 在 `Step` 开始的时候初始化，在 `Step` 结束的时候销毁 `Spring Bean`，将 `Spring Bean` 的生命周期与 `Step` 绑定。

在 `Spring Batch` 框架中，`Step Scope` 会自动被注册到 `Spring` 上下文中，如果没有使用 `Spring` 的配置文件，需要显示的声明 `Step Scope`。显示声明 `Step Scope` 参见代码清单 4-11。

代码清单 4-11 显示声明 `Step Scope`

```
1. <bean class="org.springframework.batch.core.scope.StepScope"/>
```

使用 `Step Scope` 的示例代码参见代码清单 4-12。

代码清单 4-12 使用 `Scope` 属性声明 `Bean`

```

1.     <bean:bean id="csvItemReader"
2.         class="org.springframework.batch.item.file.FlatFileItemReader"
3.         scope="step">

```

```

4.      .....
5.      </bean:bean>

```

其中，3 行：通过属性 Scope="Step"来定义 csvItemReader 的生命周期和 Step 绑定。
通过使用 Step Scope，可以支持属性的 Late Binding（属性后绑定）能力。

4.2.2 属性 Late Binding

前面对账单的示例代码中对于读取文件的配置，都是将读取文件名字在开发期直接配置在 Spring 配置文件中。考虑一下实际的对账单场景，对账单的文件通常都是根据日期自动生成的，也就是说在开发期无法知道配置文件的名字，只有在运行期才能知道配置文件的名字。幸运的是 Spring Batch 框架可以通过属性后绑定的技术，支持在运行期获取属性的值。

Spring Batch 框架通过特定的表达式支持为 Job 或者 Step 关联的实体使用后绑定技术。在 Step Scope 中 Spring Batch 框架提供的可以使用的实体包括 jobParameters、jobExecutionContext、stepExecutionContext，具体参见表 4-5。

表 4-5 Late Binding 支持的实体

实 体	说 明
jobParameters	作业参数
jobExecutionContext	当前 Job 的执行器上下文
stepExecutionContext	当前 Step 的执行器上下文

下面的例子通过使用 jobParameters 为输入的对账单文件使用后绑定技术。

前面所有的例子中，需要读取的对账单文件名全部是在配置文件中硬编码的，在实际的使用场景中是不可能确定每期账单文件名字，而只有在调用期间才知道具体的文件名，下面的例子实现了在运行期指定需要执行的文件名的功能。

在配置文件硬编码文件名的示例代码参见代码清单 4-13。

代码清单 4-13 硬编码输入文件

```

1.      <bean:bean id="csvItemReader"
2.          class="org.springframework.batch.item.file.FlatFileItemReader"
3.          scope="step">
4.          <bean:property name="resource"
5.              value="classpath:ch04/data/credit-card-bill-201303.csv"/>
6.          .....
7.      </bean:bean>

```

其中，5 行：开发期直接使用指定的文件名，不友好。

使用后绑定技术，根据输入的作业参数指定代执行的文件名。示例代码参见代码清单 4-14。

代码清单 4-14 使用 Late Binding 指定输入文件

```

1.     <bean:bean id="csvItemReader"
2.         class="org.springframework.batch.item.file.FlatFileItemReader"
3.         scope="step">
4.     <bean:property name="resource"
5.         value="#{jobParameters['inputResource']}" />
6.         .....
7.     </bean:bean>

```

其中，5 行：定义输入文件使用运行 Job 时候输入的参数"inputResource"。

调用 Job 的示例代码参见代码清单 4-15。

代码清单 4-15 执行后绑定 Job

```

1.     executeJob("ch04/job/job-stepscope.xml", "billJob",
2.         new JobParametersBuilder().addDate("date", new Date())
3.         .addString("inputResource",
4.             "classpath:ch04/data/credit-card-bill-201303.csv"));

```

其中，3~4 行：设置参数 inputResource 的值，可以在 Scope 为 Step 的 csvItemReader 中使用。

通过使用后绑定技术，避免在配置文件中使用硬编码的方式指定读取的配置文件，可以直接在运行期使用 Job 传入的参数。图 4-6 展示了 Job 配置与 Job Parameters 之间的关系。

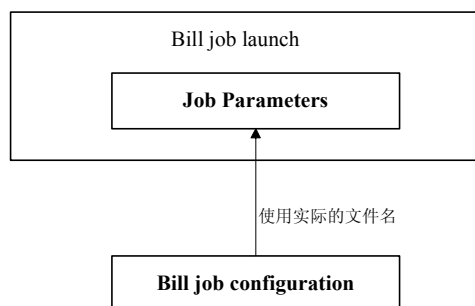


图 4-6 Job 配置与 Job Parameters 关系

4.3 运行 Job

Spring Batch 框架提供一组执行 Job 的接口 API。包括 JobLauncher、JobExplorer 和 JobOperator 三个操作 Job 的接口，三者关系参见图 4-7。JobLauncher 是最常用的作业调度器，通过给定的 Job Name 和 Job Parameters 可以执行 Job；JobExplorer 主要负责从 JobRepository 中获取执行的信息，包括获取作业实例、获取作业执行器、获取作业步执行器、获取正在运行的作业执行器、获取作业列表等操作；JobOperator 包含了 JobLauncher 和 JobExplorer 中的大部分操作。JobLauncher、JobExplorer 和 JobOperator 三者的详细描述参见表 4-6。

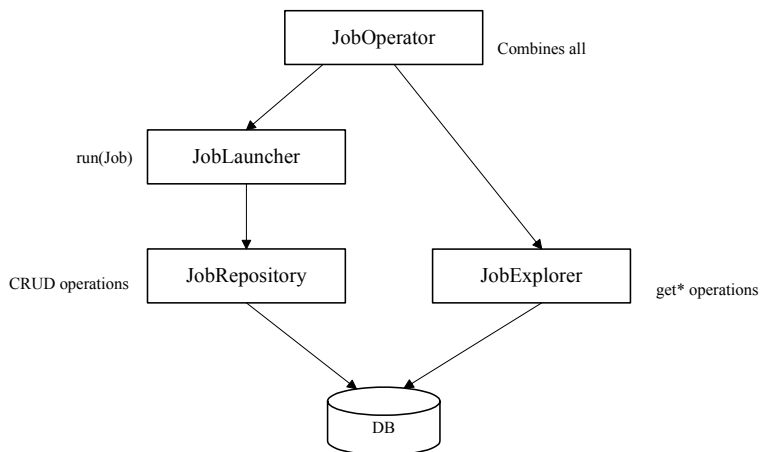


图 4-7 作业执行类 JobOperator、JobLauncher 和 JobExplorer 的关系

表 4-6 JobLauncher、JobExplorer 和 JobOperator 的详细描述

操作接口	说 明
org.springframework.batch.core.launch.JobLauncher	执行作业类
org.springframework.batch.core.explore.JobExplorer	作业状态查询类 返回作业状态的复杂对象，如 Job Execution、Job Instance、Step Execution 等
org.springframework.batch.core.launch.JobOperator	作业状态查询、作业执行类 返回作业状态的简单类型，如 Job Execution 的 id，Job Instance 的 id，Step Execution 的 id 等

JobLauncher

JobLauncher 是 Spring Batch 框架基础设施层提供的运行 Job 的能力。通过给定的 Job 名称和作业参数 JobParameters，可以通过 JobLauncher 执行 Job。通过 JobLauncher 执行 Job 时，Job 执行的状态信息通过 JobRepository 持久到数据库中。JobLauncher 接口主要操作参见图 4-8。

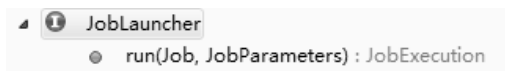


图 4-8 JobLauncher 接口操作的定义

JobExplorer

JobExplorer 主要负责从 JobRepository 中获取执行的信息，包括获取作业实例、获取作业执行器、获取作业步执行器、获取正在运行的作业执行器、获取作业列表等操作。JobExplorer 接口主要操作参见图 4-9。

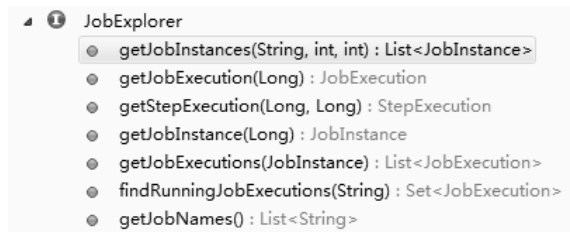


图 4-9 JobExplorer 接口操作的定义

JobOperator

JobOperator 包含了 JobLauncher 和 JobExplorer 中的大部分操作,即包括从 JobRepository 中查询作业状态信息,同时包含执行 Job 的操作。JobOperator 接口主要操作参见图 4-10。

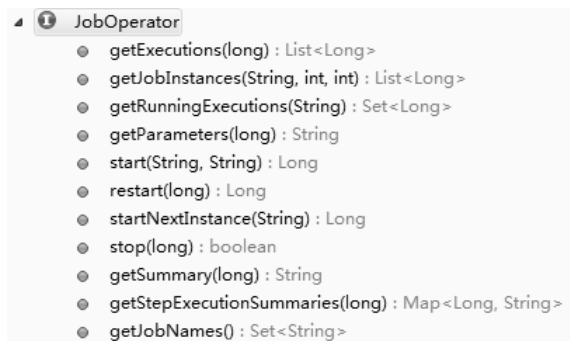


图 4-10 JobOperator 接口操作的定义

4.3.1 调度作业

前面章节我们已经介绍了如何配置 Job,一旦我们配置好 Job 后,我们需要执行 Job 作业,接下来我们描述外部系统如何通过 JobLauncher 来调度作业。代码清单 4-16 片段描述了如何通过 JobLauncher 来调度作业。

代码清单 4-16 JobLauncher 调度 Job

```

1. ApplicationContext context = new ClassPathXmlApplicationContext(
   "ch02/job/job.xml");
2. JobLauncher launcher = (JobLauncher) context.getBean("jobLauncher");
3. Job job = (Job) context.getBean("billJob");
4. try {
5.     JobExecution result = launcher.run(job, new JobParameters());
6.     System.out.println(result.toString());
7. } catch (Exception e) {
8.     e.printStackTrace();
9. }

```

JobLauncher 支持对作业的同步、异步两种调用模式。除了我们上面通过 Java Main 操作调用 Job 外，我们还可以在命令行、Web 应用、定时任务等入口调用 Job。接下来的章节我们向读者介绍 Job 的同步异步调用，以及 Job 与外界系统的调用关系。

4.3.1.1 同步异步

默认情况下，JobLauncher 的 run 操作通过同步方式调用 Job，任何调用 Job 的客户端需要等待 Job 的执行结果返回后才能结束。

同步执行 Job 作业的序列图参见图 4-11。

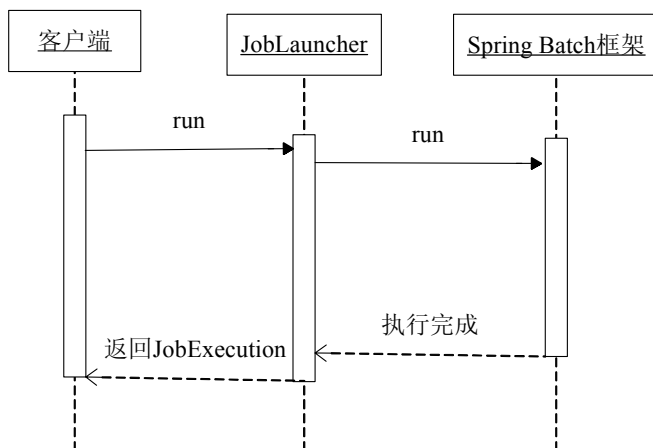


图 4-11 Job 同步执行序列图

同步操作的优势在于作业一旦执行完毕，调用客户端能够立刻收到返回值。但在实际的 Web 应用中进行 Job 调度时，因为 Job 的作业时间太长，客户端不能一直等待，用户的忍耐时间有限。同时如果 Job 作业执行时间太长，会导致 Web 容器中的大量线程被批处理作业 hold 住，无法为其他类型的请求服务，严重影响服务器的性能和吞吐量。为了解决此问题，需要通过异步的方式调用作业。如果客户端需要知道作业的执行结果，建议经过一段时间后，主动查询当前作业的执行情况。JobLauncher 提供了异步执行 Job 的能力。

异步执行 Job 作业的序列图参见图 4-12。

配置异步调用的 JobLauncher 只需要增加属性 taskExecutor，该属性表示当前执行的线程池，配置代码参见代码清单 4-17。

代码清单 4-17 配置异步 JobLauncher

```
1. <task:executor id="executor" pool-size="5" />
2. <!-- 异步作业调度器 -->
3. <bean:bean id="jobLauncherAsyn"
4.     class="org.springframework.batch.core.launch.support.
       SimpleJobLauncher">
```

```

5.      <bean:property name="jobRepository" ref="jobRepository"/>
6.      <bean:property name="taskExecutor" ref="executor" />
7.      </bean:bean>

```

其中，1 行：配置大小为 5 的线程池。

6 行：为作业调度器配置执行的线程池，作业执行期间会从线程池选择一个线程执行 Job。

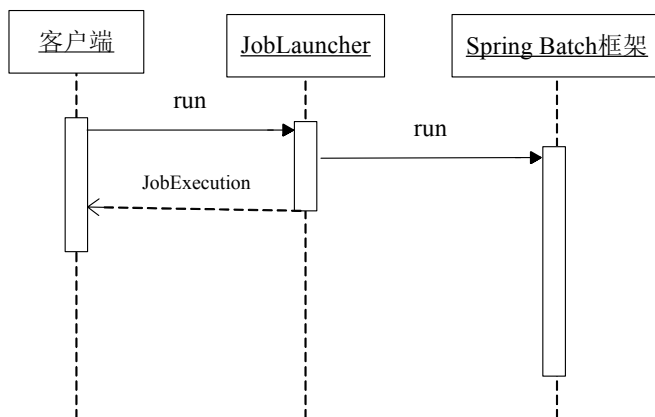


图 4-12 Job 异步执行序列图

运行代码工程中的 `test.com.juxtapose.example.ch04.JobLaunchAsyn`，仔细观察控制台输出：首先会输出 Job Execution 的信息，然后输出 Job 执行的具体信息。即在 JobLauncher 的 run 方法被调用后，客户端立刻返回；同时后台使用其他线程执行 Job 作业。

4.3.1.2 Job 与外界系统

在实际的 Job 使用场景中，标准 Web 应用、定时任务调度器、命令行等都可能触发不同的 Job 操作。图 4-13 展示了批处理作业和外界系统的关系图。

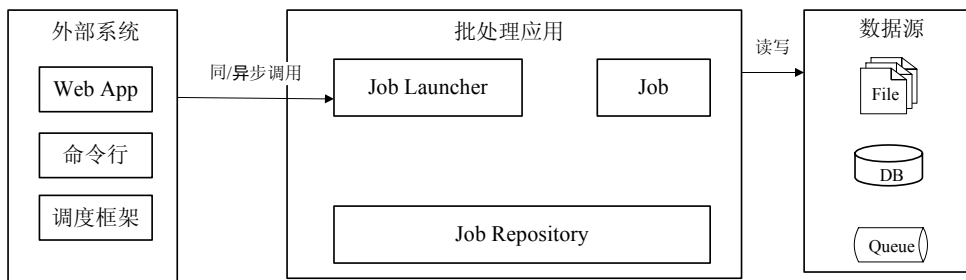


图 4-13 批处理作业和外界系统的关系图

接下来的三节我们将学习如何通过命令行、调度框架和 Web 应用程序执行 Job 作业。

4.3.2 命令行执行

Spring Batch 框架提供通过命令行方式执行 Job 的入口, 通过命令行的方式可以在一个单独的 JVM 中执行批处理作业。通过命令行的方式可以手动触发, 也可以定义自动任务通过脚本的方式执行批处理作业。Spring Batch 框架提供的命令行执行类: `org.springframework.batch.core.launch.support.CommandLineJobRunner`。

通过命令行方式执行 Job 的关系参见图 4-14。

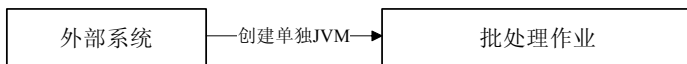


图 4-14 通过命令行执行批处理作业

示例

按照下面的步骤通过命令行执行 Hello World 章中的批处理作业。

(1) 构建示例项目 `spring-batch-example`。

可以在 Eclipse 中通过 Maven 插件构建项目, 在 `target` 目录中生成 jar 文件。

(2) 调出命令行, 进入目录: `/spring-batch-example/target`。

(3) 执行如下的命令, 命令参见代码清单 4-18。

代码清单 4-18 命令行执行 Job 命令

```
java -classpath "./dependency/*;spring-batch-example-1.0.jar"
    org.springframework.batch.core.launch.support.CommandLineJobRunner
    ch02/job/job.xml
    billJob
```

命令参数说明:

`-classpath "./dependency/*;spring-batch-example-1.0.jar"` 表示当前执行的 classpath。

`ch02/job/job.xml` 表示执行的 Spring Batch 的配置文件。

`billJob` 表示需要执行的批处理作业。

(4) 查看控制台, 会成功打印出执行的文件信息, 控制台内容输出参见代码清单 4-19。

代码清单 4-19 命令行执行 Job 的控制台输出

```
1. 2013-03-18 20:29:52,397 INFO [org.springframework.batch.core.job.
   SimpleStepHandler] - Executing step: [billStep]
2. accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
   12:00:08;address=Lu Jia Zui road
3. accountID=4047390012345678;name=tom;amount=320.0;date=2013-2-3
   10:35:21;address=Lu Jia Zui road
4. accountID=4047390012345678;name=tom;amount=674.7;date=2013-2-6
   16:26:49;address=South Linyi road
5. accountID=4047390012345678;name=tom;amount=793.2;date=2013-2-9
   15:15:37;address=Longyang road
```

```

6.  accountID=4047390012345678;name=tom;amount=360.0;date=2013-2-11
    11:12:38;address=Longyang road
7.  accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28
    20:34:19;address=Hunan road

```

(5) 查看文件/spring-batch-example/target/target/ch02/outputFile.csv，内容会正确地输出。
其中，CommandLineJobRunner 参数说明如下。

```

java -classpath "xxx" CommandLineJobRunner jobPath <options> jobIdentifier
(jobParameters)

```

jobPath: CommandLineJobRunner 默认使用 ClassPathXmlApplicationContext 从当前的 classpath 中加载配置文件，可以通过写前缀的方式更改 CommandLineJobRunner 的加载器，例如使用 file:./job.xml 方式，表示使用当前目录下面的 job.xml 文件。

options: 可选的参数，支持的参数包括-restart、-stop、-abandon、-next。详细说明参见表 4-7。

表 4-7 CommandLineJobRunner 命令 options 参数说明

可选参数	说 明
-restart	根据 jobIdentifier 重启最后一次失败的作业
-stop	根据 jobIdentifier 停止正在执行的作业
-abandon	根据 jobIdentifier 废弃 stopped 的作业
-next	根据 JobParametersIncrementer 执行下一个作业

jobIdentifier: 作业的名字（正常执行一个 Job 的时候，需要传入 Job 的名字，即 Spring 配置文件中配置 Job 的 Bean ID），或者是 job execution 的 id（当使用 -restart、-stop、-abandon 时候，需要传入 job execution 的 id）。

job Parameters: 作业参数可以有 0 到多个，其支持的类型为 String、Date、Long、Double。Job Parameters 支持的参数及示例参见表 4-8。

表 4-8 Job Parameters 支持的参数及示例

参数类型	示 例
String	inputResource(string)=/ch02/job.xml
Date	createTime(date)=2013/03/18
Long	timeout(long)=5000
Double	account(double)=100.23

获取作业退出状态

通过 CommandLineJobRunner 在命令行中执行 Job 同样可以获取 Job 执行的退出状态 ExitStatus，默认情况 Job 的退出状态 ExitStatus 和 Job 的批处理状态 BatchStatus 一致。CommandLineJobRunner 的退出状态值由类 org.springframework.batch.core.launch.support.SimpleJvmExitCodeMapper（实现接口：org.springframework.batch.core.launch.support.ExitCodeMapper）

定义，具体返回的退出状态值参见表 4-9。

表 4-9 命令行执行 Job 的退出状态值

状 态 值	说 明
0	作业正常完成，状态为 COMPLETED
1	作业完成失败，状态为 FAILED
2	作业失败，例如没有此作业

可以通过重新实现接口 `org.springframework.batch.core.launch.support.ExitCodeMapper`，来自定义作业的退出状态。例如通过自定义退出状态类，可以实现正常完成状态退出为 1，失败为 2，停止为 3，其他为 4。代码清单 4-20 为自定义退出状态实现类代码。

代码清单 4-20 自定义退出状态实现类

```
1. public class CustomerExitCodeMapper implements ExitCodeMapper {
2.     public int intValue(String exitCode) {
3.         if (ExitStatus.COMPLETED.getExitCode().equals(exitCode)) {
4.             return 1;
5.         } else if (ExitStatus.FAILED.getExitCode().equals(exitCode)) {
6.             return 2;
7.         } else if (ExitStatus.STOPPED.getExitCode().equals(exitCode)) {
8.             return 3;
9.         } else {
10.            return 4;
11.        }
12.    }
13. }
```

为了能在命令行中使用上面开发的退出码匹配类，需要在配置文件中定义上面的实现，配置内容参见代码清单 4-21。

代码清单 4-21 配置自定义退出码实现类

```
1. <bean:bean class="com.juxtapose.example.ch04.exitstatus.Customer
   ExitCodeMapper">
2. </bean:bean>
```

可以通过代码清单 4-22 中的命令行，执行 `job-exitstatus.xml` 中定义的 `billJob` 作业。

代码清单 4-22 命令行执行 Job，使用自定义退出码实现类

```
java -classpath "./dependency/*;spring-batch-example-1.0.jar"
    org.springframework.batch.core.launch.support.CommandLineJobRunner
    ch04/job/job-exitstatus.xml
    billJob
    inputResource(string)=ch04/data/credit-card-bill-201303.csv
    outputResource(string)=file:target/ch04/exitstatus/outputFile.csv
    date(string)=2013/03/18
```

说明：在使用 `CommandLineJobRunner` 进行命令行执行过程中，Spring 对 `CommandLineJobRunner` 中的依赖全部使用基于类型的自动装配。例如 `CommandLineJobRunner` 中的属性 `launcher`，Spring 会自动基于类型 `JobLauncher` 进行装配，如果没有 `JobLauncher` 的定义或者有多个定义都会导致错误。

代码清单 4-23 展示当有 2 个 `JobLauncher` 定义时候的错误信息。

代码清单 4-23 命令行执行 Job 存在多个 `JobLauncher` 时，错误输出

```
: No unique bean of type [org.springframework.batch.core.launch.JobLauncher]
is defined: expected single matching bean but found 2: [jobLauncher,
jobLauncherAsync]
    at org.springframework.beans.factory.support.AbstractAutowireCapable
BeanFactory.autowireByType (AbstractAutowireCapableBeanFactory.java:1167)
.....
```

4.3.3 与定时任务集成

Spring Batch 提供了 Job 的执行能力，其本身不是一个定时的调度框架，因此可以将定时调度框架和 Spring Batch 结合起来完成定时作业。Spring 本身提供了一个轻量级的调度框架 `Spring scheduler`。本节展示如何通过 `Spring scheduler` 定时调度 Spring Batch 中的 Job。

图 4-15 展示了 Spring Batch 框架和 `Spring scheduler` 之间的关系。

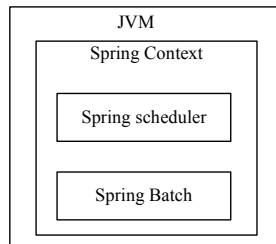


图 4-15 Spring Batch 框架和 `Spring scheduler` 关系图

`Spring scheduler` 使用非常简单，只需要完成 2 步即可完成简单的定时调度功能。

- (1) 定义一个 `scheduler`，该 `scheduler` 提供执行定时任务的线程。
- (2) 定义需要定时操作的方法和调度周期。

`Spring Scheduler` 和 `Spring Batch Job` 的配置参见代码清单 4-24。完整代码参见文件 `ch04/job/job-spring-scheduler.xml`。

代码清单 4-24 配置 `Spring Scheduler` 和 `Spring Batch Job`

```
1. <task:scheduler id="scheduler" pool-size="10" />
2.
3. <!-- 每一秒钟，执行对象 schedulerLauncher 的 launch 方法一次 -->
4. <task:scheduled-tasks scheduler="scheduler">
```



```

5.         <task:scheduled ref="schedulerLauncher" method="launch" fixed-
           rate="1000" />
6.     </task:scheduled-tasks>
7.
8.     <bean:bean id="schedulerLauncher"
9.         class="com.juxtapose.example.ch04.scheduler.SchedulerLauncher">
10.        <bean:property name="job" ref="helloworldJob" />
11.        <bean:property name="jobLauncher" ref="jobLauncher" />
12.    </bean:bean>
13.
14.    <!-- HelloWorld Job -->
15.    <job id="helloworldJob">
16.        <step id="helloworldStep">
17.            <tasklet>
18.                <bean:bean class="com.juxtapose.example.ch04.HelloWorld
19.                    Tasklet">
20.                </bean:bean>
21.            </tasklet>
22.        </step>
23.    </job>

```

该程序说明如下。

1 行：定义执行定时任务的线程池大小。

4~6 行：fixed-rate="1000"表示每秒执行一次对象 schedulerLauncher 的 launch 方法。

8~12 行：定时任务执行的对象声明，在类 com.juxtapose.example.ch04.scheduler.SchedulerLauncher 中的 launch 方法负责具体 Job 的调用。

15~22 行：定义了自定义实现 Tasklet 的任务。

com.juxtapose.example.ch04.scheduler.SchedulerLauncher 类的实现参见代码清单 4-25。

代码清单 4-25 定时调度类 SchedulerLauncher 定义

```

1. public class SchedulerLauncher {
2.     private Job job;
3.     private JobLauncher jobLauncher;
4.
5.     public void launch() throws Exception {
6.         JobParameters jobParams = createJobParameters();
7.         jobLauncher.run(job, jobParams);
8.     }
9.
10.    private JobParameters createJobParameters() {
11.        JobParameters jobParams = new JobParametersBuilder().
12.            addDate("executeDate", new Date()).toJobParameters();
13.        return jobParams;
14.    }

```

```
15.      .....省略 get* set*方法
16. }
```

其中，5~8行：执行具体的 job 调度。

执行测试类 `test.com.juxtapose.example.ch04.JobLaunchScheduler`，控制台每一秒钟调度一次任务，控制台输出信息参见代码清单 4-26。

代码清单 4-26 定时调度 Job 的控制台输出

```
Execute job :helloworldJob.
JobParameters:executeDate = Tue Mar 19 20:20:53 CST 2013 (DATE);
.....
Execute job :helloworldJob.
JobParameters:executeDate = Tue Mar 19 20:20:54 CST 2013 (DATE);
.....
Execute job :helloworldJob.
JobParameters:executeDate = Tue Mar 19 20:20:55 CST 2013 (DATE);
.....
Execute job :helloworldJob.
JobParameters:executeDate = Tue Mar 19 20:20:56 CST 2013 (DATE);
```

注意：……处表示省略了其他日志信息。

通过本节学习，可以将 Spring Batch 框架和其他的调度框架进行集成使用，如 Quartz 框架、Cron（Linux 环境下的定时任务工具）。

4.3.4 与 Web 应用集成

Spring Batch 框架基于 Spring 开发，可以方便地内嵌在 Web 应用中使用，这样批处理作业可以通过 HTTP 协议进行远程的访问。同样可以在 Web 应用中内嵌定时任务处理框架，方便在 Web 应用内部通过定时框架调用 Spring Batch 中定义的 Job。

Web 应用、Spring Batch 框架和 Spring scheduler 的关系参见图 4-16。

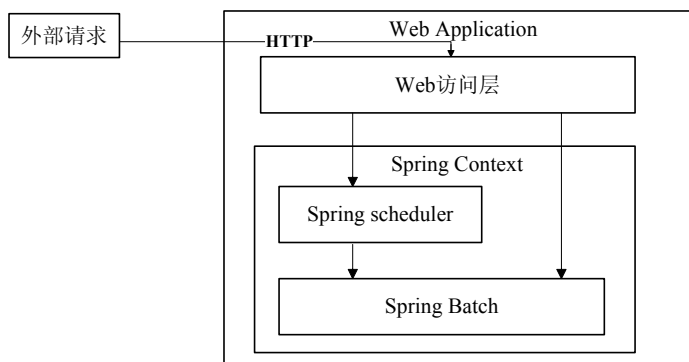


图 4-16 Web 应用、Spring Batch 框架和 Spring scheduler 关系图

外部请求可以直接通过 Web 访问层执行批处理作业，也可以由 Web 访问层访问批处理框架，然后由批处理框架调度批处理作业。

使用 Spring MVC 框架，调用指定的 Spring Batch 的 Job。Spring MVC 框架通过定义一个 controller 来描述具体的 HTTP 服务，controller 可以通过简单的 annotation（@Controller、@RequestMapping、@RequestParam 等）来描述。

配置 Controller

com.juxtapose.example.ch04.web.JobLauncherController 类定义代码，参见代码清单 4-27。

代码清单 4-27 JobLauncherController 类定义

```
1. @Controller
2. public class JobLauncherController {
3.     private static final String JOB_NAME = "jobName";
4.     private JobLauncher jobLauncher;
5.     private JobRegistry jobRegistry;
6.
7.     public JobLauncherController(JobLauncher jobLauncher, JobRegistry
      jobRegistry) {
8.         this.jobLauncher = jobLauncher;
9.         this.jobRegistry = jobRegistry;
10.    }
11.
12.    @RequestMapping(value="executeJob",method=RequestMethod.GET)
13.    public void launch(@RequestParam String jobName,HttpServletRequest
      request) throws Exception
14.    {
15.        JobParameters jobParameters = bulidParameters(request);
16.        jobLauncher.run( jobRegistry.getJob(jobName),jobParameters);
17.    }
18.
19.    private JobParameters bulidParameters(HttpServletRequest request) {
20.        JobParametersBuilder builder = new JobParametersBuilder();
21.        @SuppressWarnings("unchecked")
22.        Enumeration<String> paramNames = request.getParameterNames();
23.        while(paramNames.hasMoreElements()) {
24.            String paramName = paramNames.nextElement();
25.            if(!JOB_NAME.equals(paramName)) {
26.                builder.addString(paramName,request.getParameter
                  (paramName));
27.            }
28.        }
29.        return builder.toJobParameters();
30.    }
31. }
```

该程序说明如下。

1 行: `@Controller` 表示 MVC 框架中的控制器。

12 行: `@RequestMapping (value="executeJob",method=RequestMethod.GET)` 表示当前操作对应请求为 `executeJob`, 通过 `get` 的方式访问。

13 行: `@RequestParam String jobName` 表示 `jobName` 为请求的参数, 表示需要执行 Job 的名字。

16 行: 执行具体的 Job, Job 具体的对象通过 `jobRegistry` 获取 (`jobRegistry` 中存放 Spring 配置文件中定义的所有 Job 信息, 具体配置参见后面的作业配置文件)。

19~30 行: 将 HTTP 请求的参数转换为作业参数。

配置 MVC 框架

在 `web.xml` 中配置加载 Spring 容器的 listener 和对应的 servlet 配置。`web.xml` 配置定义参见代码清单 4-28。

代码清单 4-28 Web 应用调用 Job 的 `web.xml` 配置

```
1.    <display-name>SpringBatch</display-name>
2.    <listener>
3.        <listener-class>org.springframework.web.context.Context
        LoaderListener
4.    </listener-class>
5.    </listener>
6.    <servlet>
7.        <servlet-name>batchapp</servlet-name>
8.        <servlet-class>org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
9.    </servlet>
10.   <servlet-mapping>
11.       <servlet-name>batchapp</servlet-name>
12.       <url-pattern>/*</url-pattern>
13.   </servlet-mapping>
```

其中, 3 行: 定义加载 Spring 容器, 默认加载 `WEB-INF/applicationContext.xml` 来初始化 Spring 容器。

6~10 行: 定义匹配的 url, MVC 框架默认使用 (`servlet-name`) -`servlet.xml` 文件来加载控制器, 本例子中加载文件 `batchapp-servlet.xml`。配置控制器文件 `batchapp-servlet.xml` 参见代码清单 4-29。

代码清单 4-29 配置控制器文件 `batchapp-servlet.xml`

```
1.    <bean class="com.juxtapose.example.ch04.web.JobLauncherController">
2.        <constructor-arg ref="jobLauncher" />
3.        <constructor-arg ref="jobRegistry" />
4.    </bean>
```

其中，1~4 行：定义了对应的 web controller。

需要注意 `JobLauncherController` 中使用了两个对象：`jobLauncher` 和 `jobRegistry`，前者负责调用 `Job` 执行；后者定义了 `Job` 的注册器，`jobRegistry` 存放了所有的 `Job` 对象，可以根据 `Job` 的名字获取对应的 `Job` 对象。

`JobLauncherController` 中引用的 `jobLauncher` 和 `jobRegistry` 从 `ContextLoaderListener` 中加载的 `Spring` 容器获取对象。`ContextLoaderListener` 默认加载文件 `applicationContext.xml` 来初始化 `Spring` 容器。

配置 Spring 容器初始化文件

`applicationContext.xml` 负责引用所有的 `Spring` 配置文件，可以看作加载 `Spring` 容器的入口。本例中 `applicationContext.xml` 引入了 `job-context.xml` 和 `job-sample.xml` 文件。`applicationContext.xml` 文件的定义参见代码清单 4-30。

代码清单 4-30 `applicationContext.xml` 文件定义

```
1.      <import resource="job-context.xml"/>
2.      <import resource="job-sample.xml"/>
```

`job-context.xml` 中定义基本的批处理框架定义。

`job-sample.xml` 文件定义了一个示例的 `Job`。

`JobLauncherController` 用到的 `jobLauncher` 和 `jobRegistry` 的定义参见代码清单 4-31。

代码清单 4-31 `JobLauncherController` 类定义

```
1.      <bean:bean class="org.springframework.batch.core.configuration.support.
2.                                     JobRegistryBeanPostProcessor">
3.          <bean:property name="jobRegistry" ref="jobRegistry" />
4.      </bean:bean>
5.
6.      <bean:bean id="jobRegistry"
7.          class="org.springframework.batch.core.configuration.support.
              MapJobRegistry" />
```

其中，1~4 行：通过类 `JobRegistryBeanPostProcessor` 的声明，可以在 `Job` 定义加载后，自动注册到 `jobRegistry` 中。

部署应用

将示例工程目录 `spring-batch-example/src/main/resources/ch04/webapp/` 目录下的内容复制到 `tomcat` 目录 `webapps` 下面，应用名为 `batchapp`。

应用 `batchapp` 目录结构参见图 4-17。

应用部署后，启动 `tomcat` 应用，

如下的 url 访问，通过 HTTP 请求执行作业 `chunkJob`，使用的作业参数为 `date`。

`http://localhost:8080/batchapp/executeJob?jobName=chunkJob&date=20130320`

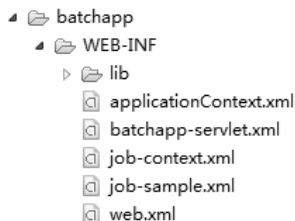


图 4-17 batchapp 目录结构

4.3.5 停止 Job

批处理作业执行过程中，如果发生了意外需要终止当前执行的 Job，Spring Batch 框架提供了停止正在运行 Job 的能力，通过接口 `org.springframework.batch.core.launch.JobOperator` 中的 `stop` 方法来实现；另外在作业执行过程中，开发人员可以根据业务的需要终止正在运行的 Job，可以在 Job 的业务代码中定义何时终止 Job。接下来我们学习这两种停止 Job 的能力。

4.3.5.1 JobOperator 停止 Job

Job 执行期间通过 `org.springframework.batch.core.launch.JobOperator` 的 `stop` 方法停止正在运行的 Job。`stop` 方法返回 `boolean` 值，表示当前终止消息是否成功发送，至于什么时候终止消息，则由 Spring Batch 框架决定。

图 4-18 展示了 `JobOperator` 的操作定义。

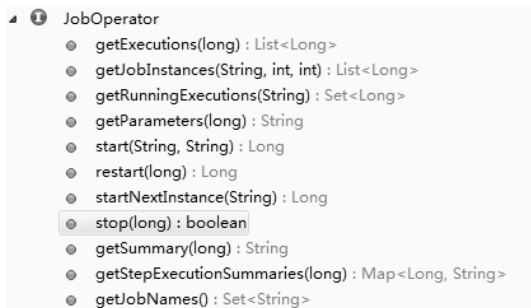


图 4-18 JobOperator 定义

可通过 `JobOperator` 的 `stop` 方法停止正在运行的 Job，代码参见代码清单 4-32。

代码清单 4-32 通过 JobOperator 的 stop 操作停止 Job

```
1. Set<Long> runningExecutions = jobOperator.getRunningExecutions(jobName);
2. Iterator<Long> iterator = runningExecutions.iterator();
3. while(iterator.hasNext()) {
4.     boolean sendMessage = jobOperator.stop(iterator.next());
5.     System.out.println("sendMessage:" + sendMessage);
6. }
```

其中，1 行：根据 `jobName` 查找正在运行的所有的作业执行器的 ID。

3~6 行：迭代执行，停止所有正在运行的 Job。

4 行：根据具体的作业执行器的 ID 停止 Job。

查看类 `org.springframework.batch.core.launch.JobOperator` 的源代码，`stop` 方法调用后并没有立刻终止 Job，而是向 Spring Batch 框架发送了终止请求，对应的返回值为 `true` 表示当前终止作业的消息已经成功发送，如果对应的返回值为 `false` 表示终止作业的消息发送失败。只有当停止消息发送成功后，并且作业处理进入到 Spring Batch 框架时，才会真正停止正在运行的 Job。如果业务操作一直运行，会导致停止 Job 会需要很多时间。

接下来通过一个完整的示例，来展示如何通过 `JobOperator` 的 `stop` 方法来终止 Job。

配置文件

上面代码完成后，需要在配置文件中定义 `JobOperator` 及对应的属性定义，配置内容参见代码清单 4-33。

完整配置文件参见 `ch04/job/job-stop.xml`。

代码清单 4-33 配置 `JobOperator` 及属性

```
1. <job id="chunkJob">
2.     <step id="chunkStep">
3.         <tasklet>
4.             <chunk reader="reader" writer="writer" commit-interval="10" />
5.         </tasklet>
6.     </step>
7. </job>
8. <bean:bean
9.     class="org.springframework.batch.core.configuration.support.
10.         JobRegistryBeanPostProcessor">
11.     <bean:property name="jobRegistry" ref="jobRegistry" />
12. </bean:bean>
13.
14. <bean:bean id="jobRegistry"
15.     class="org.springframework.batch.core.configuration.support.
16.         MapJobRegistry" />
17.
18. <bean:bean id="jobExplorer"
19.     class="org.springframework.batch.core.explore.support.
20.         JobExplorerFactoryBean">
21.     <bean:property name="dataSource" ref="dataSource" />
22. </bean:bean>
23.
24. <bean:bean id="jobOperator"
25.     class="org.springframework.batch.core.launch.support.
```

```

SimpleJobOperator">
24.     <bean:property name="jobRepository" ref="jobRepository" />
25.     <bean:property name="jobLauncher" ref="jobLauncherAsyn" />
26.     <bean:property name="jobRegistry" ref="jobRegistry" />
27.     <bean:property name="jobExplorer" ref="jobExplorer" />
28. </bean:bean>

```

该程序说明如下。

1~7 行：定义作业，作业的具体 reader 和 writer 不在此贴出，可以到示例工程代码中查看，reader 功能是从 0 开始读取数据，直到 Integer 的最大值；writer 将读到的数据打印在控制台上。

8~12 行：定义 Job 自动注册功能。

14~15 行：定义作业注册器。

17~20 行：定义作业的浏览接口，可以通过数据库查询 Job 执行的元数据信息。

22~28 行：定义我们需要的 jobOperator，可以对作业实例进行 CRUD 和控制处理。

18 行：使用了异步的作业调度器：**jobLauncherAsyn**，目的是后面测试代码中可以在同一个线程中方便终止 Job。

示例代码

在本示例代码中，首先通过异步的作业调度器来执行一个 Chunk 类型的作业，然后通过 JobOperator 终止 Job。示例代码参见代码清单 4-34。

完整的示例代码参见工程中的类：`test.com.juxtapose.example.ch04.JobLaunchStop`。

代码清单 4-34 异步作业调度器执行 Job，JobOperator 终止 Job 示例代码

```

1. public static void executeJobAndStop(String jobPath, String jobName,
2.     JobParametersBuilder builder) {
3.     ApplicationContext context = new ClassPathXmlApplicationContext
4.         (jobPath);
5.     JobLauncher launcher = (JobLauncher) context.getBean("jobLauncherAsyn");
6.     JobOperator jobOperator = (JobOperator) context.getBean
7.         ("jobOperator");
8.     Job job = (Job) context.getBean(jobName);
9.     try {
10.         launcher.run(job, builder.toJobParameters());
11.         Set<Long> runningExecutions = jobOperator.getRunningExecutions
12.             (jobName);
13.         Iterator<Long> iterator = runningExecutions.iterator();
14.         while(iterator.hasNext()){
15.             boolean sendMessage = jobOperator.stop(iterator.next());
16.             System.out.println("sendMessage:" + sendMessage);
17.         }
18.     } catch (Exception e) {
19.         e.printStackTrace();
20.     }
21. }

```



```

17.     }
18. }
19.
20. /**
21.  * @param args
22.  */
23. public static void main(String[] args) {
24.     executeJobAndStop("ch04/job/job-stop.xml", "chunkJob",
25.         new JobParametersBuilder().addDate("date", new Date()));
26. }

```

其中，9 行：通过 `jobOperator` 查找作业名“`chunkJob`”的所有正在运行的作业执行器 ID。
 12 行：根据作业执行器 ID，发送停止作业的消息。
 24～25 行：方法入口，使用配置文件 `ch04/job/job-stop.xml` 中的定义，启动并终止 `chunkJob`。

运行用例

执行用例 `test.com.juxtapose.example.ch04.JobLaunchStop`，控制台信息可以看到仅执行了一次循环操作。

代码清单 4-35 展示了控制台输出的信息。

代码清单 4-35 Job 执行控制台输出

```

2013-03-20    07:57:28,147    INFO    [org.springframework.batch.core.job.
SimpleStepHandler] - Executing step: [chunkStep]
sendMessage:true
Write begin:
1,2,3,4,5,6,7,8,9,10,Write end!!

```

通过控制台信息可以看出，作业步仅执行了一次循环就停止了。为了验证我们的猜测，查看作业步执行器表，可以看出读的次数仅有 10 次，正好验证了我们根据控制台信息的猜测。

表 `batch_step_execution` 中的数据参见图 4-19。读操作一共有 10 次。

STEP_EXECUTION_ID	STEP_NAME	READ_COUNT	JOB_EXECUTION_ID	VERSION
114	chunkStep	10	115	3

图 4-19 作业步执行器记录

到对应的数据库中查看作业的执行状态，首先查看作业实例表，`chunkJob` 新产生作业实例编号为 114，再查看作业执行器表，对应作业实例 ID 为 114 的作业执行器的状态为 `STOPPED` 状态。

作业实例表 `batch_job_instance` 中的数据，参见图 4-20。

JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
114	0	chunkJob	70c3852431377e55179947a7dbb6e251

图 4-20 作业实例记录

作业执行器表 `batch_job_execution` 中的数据，参见图 4-21。

JOB_EXECUTION_ID	STATUS	EXIT_CODE	JOB_INSTANCE_ID	VERSION
115	STOPPED	STOPPED	114	3

图 4-21 作业执行器记录

通过 JMX 方式操作 JobOperator

上面的示例演示了如何在 Java 代码中执行停止作业，Spring 框架同样提供了其他的操作方式来友好地终止正在执行的作业。Spring 框架同样提供友好的方式将 Bean 暴露为 JMX 服务，仅仅需要在 Spring 配置文件中简单地定义。JConsole 提供了简单的执行 JMX 服务的入口，一旦 JobOperator 定义为 JMX 服务后，就可以通过 JConsole 的方式操作对应的 stop 操作。

将 JobOperator 配置为 JMX 服务参见代码清单 4-36。

代码清单 4-36 JobOperator 配置为 JMX 服务

```
1. <bean:bean class="org.springframework.jmx.export.MBeanExporter">
2.     <bean:property name="beans">
3.         <bean:map>
4.             <bean:entry key="com.juxtapose.example:name=jobOperator"
5.                 value-ref="jobOperator" />
6.         </bean:map>
7.     </bean:property>
8. </bean:bean>
```

其中，4~5 行：将对象 `jobOperator` 暴露为 JMX 服务，定义服务名称为：`"com.juxtapose.example:name=jobOperator"`。

按照下面的操作步骤执行，展示了如何通过 JConsole 方式停止正在执行的 Job。

(1) 执行类 `test.com.juxtapose.example.ch04.JobLaunchStopJMX`。

(2) 通过 DB 查看当前作业的执行器 ID，本例操作的时候为 119。

(3) 在命令行输入 `jconsole` 命令，调出 JMX 控制台，参见图 4-22。

(4) 选择名字为 `test.com.juxtapose.example.ch04.JobLaunchStopJMX` 的进程，单击连接后进入步骤 5 的监控页面，参见图 4-23。



图 4-22 JConsole 登录界面

(5) 进入控制台后，选择 MBean 页，找到对应的 stop 操作



图 4-23 JConsole 控制台界面

(6) 选择 stop 操作后，输入参数 119，单击 stop 按钮，即可以停止当前的任务操作。

执行 stop 操作的返回值为 true，表示当前停止任务的请求已经被成功发送；如果要看任务是否被正确停止，需要参照步骤 7 的方式到数据库查看数据。

stop 操作的返回值参见图 4-24。

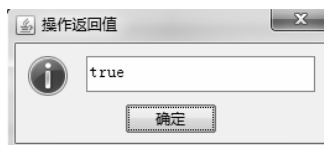


图 4-24 JConsole 操作返回值界面

(7) 查看数据库表 batch_job_execution，确认执行器为 119 的 Job 已经被停止。

图 4-25 展示了执行器为 119 的作业状态为 STOPPED。

	JOB_EXECUTION_ID	STATUS	EXIT_CODE	JOB_INSTANCE_ID	VERSION	CREATE_TIME
1	116	STOPPED	STOPPED	115	3	2013-03-20 08:23:05
2	117	STOPPED	STOPPED	116	3	2013-03-20 08:30:22
3	118	STARTED	UNKNOWN	117	1	2013-03-20 08:38:32
4	119	STOPPED	STOPPED	118	3	2013-03-20 08:38:55

图 4-25 执行器为 119 的作业状态

4.3.5.2 业务停止 Job

JobOperator 提供停止 Job 能力，通过 JobOperator 可以在作业外部，如 JMX 控制台，或者外部独立的进程中终止 Job。Spring Batch 框架同样提供了在作业步执行期间终止任务的能力。通过 StepExecution 中的 setTerminateOnly 操作可以终止正在运行的任务。

StepExecution.setTerminateOnly()会发送一个停止消息给框架，一旦 Spring Batch 框架接收到停止消息，并且框架获取作业的控制权，Spring Batch 框架会自动终止作业。

在通过业务操作终止任务操作时，不要在读、处理、写的业务逻辑中终止任务，以尽量保证业务操作的完整性。Spring Batch 框架提供了丰富的拦截器机制，可以在拦截器中执行任务的终止，例如在 ItemReadListener 中的 beforeRead()中终止任务。

通过拦截器 (com.juxtapose.example.ch04.stop.StopStepListener<T>) 终止任务示例代码参见代码清单 4-37。

代码清单 4-37 作业终止拦截器 StopStepListener

```
1. public class StopStepListener<T> implements StepExecutionListener,
   ItemReadListener<T> {
2.     private StepExecution stepExecution;
3.     private Boolean stop = Boolean.FALSE;
4.
5.     public void beforeStep(StepExecution stepExecution) {
6.         this.stepExecution = stepExecution;
7.     }
8.
9.     public void beforeRead() {
10.        if(isStop()) {
11.            this.stepExecution.setTerminateOnly();
12.        }
13.    }
14.    .....
15. }
```

其中，6 行：获取 stepExecution，并保存在拦截器的实例中。

11 行：判断是否终止任务，如果终止调用操作 stepExecution.setTerminateOnly()实现配置定义的拦截器，配置文件参见代码清单 4-38。

代码清单 4-38 作业步中配置业务停止拦截器

```
1. <job id="chunkBusinessJob">
2.     <step id="chunkBusinessStep">
3.         <tasklet>
4.             <chunk reader="reader" writer="writer" commit-interval="10" />
5.             <listeners>
6.                 <listener ref="stopListener"></listener>
7.             </listeners>
```

```
8.         </tasklet>
9.     </step>
10. </job>
11. <bean:bean id="stopListener" class="com.juxtapose.example.ch04.stop.
    StopStepListener" />
```

其中，6 行：定义作业步 `chunkBusinessStep` 的拦截器，引用 **stopListener**。

11 行：定义拦截器对象。

执行测试类 `test.com.juxtapose.example.ch04.JobLaunchStopBusiness`，可以在控制台看出任务被友好的方式停止掉。

基于拦截器的机制终止任务，可以保证业务代码专注于完成业务操作，保证了代码的优雅性。可以在拦截器 `org.springframework.batch.core.StepListener` 及子类的拦截器中进行作业的停止控制。

配置作业步 Step

Step 表示作业中的一个完整步骤，一个 Job 可以由一个或者多个 Step 组成。Step 包含了一个实际运行的批处理任务中所有必需的信息。如果忘记了 Job 和 Step 的关系，可以参考 3.3 节。Step 的声明使用 step 元素声明，每个 Step 由 tasklet 元素描述具体的作业，tasklet 可以是一个自定义的业务处理（需要实现接口 `org.springframework.batch.core.step.tasklet.Tasklet`），也可以使用 chunk 元素描述面向块的业务处理，一个典型的 chunk 包含 read、process 和 write 三个操作。

一个典型的 Step 声明参见代码清单 5-1。

代码清单 5-1 典型的 Step 声明

```
1.      <step id="billStep">
2.          <tasklet transaction-manager="transactionManager">
3.              <chunk reader="csvItemReader" writer="csvItemWriter"
4.                  processor="creditBillProcessor" commit-interval="2">
5.              </chunk>
6.          </tasklet>
7.      </step>
```

step、tasklet、chunk、read、processor、write 之间的关系如图 5-1 所示。Step 的作用域最大，然后是 Tasklet，接下来为 Chunk，在每个 Chunk 中可以定义 read、process、write。

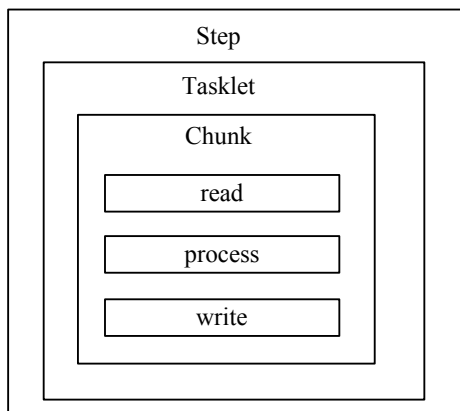


图 5-1 step、tasklet、chunk、read、processor、write 关系图

5.1 配置 Step

在配置 Step 之前，我们一起看一下 Step 的主要属性定义和元素定义。

Step 主要属性包括 id（作业步唯一标识）、next（下一个执行的作业步）、parent（指定该作业步的父作业步）、job-repository（定义作业仓库）、abstract（定义作业步是否是抽象的）。图 5-2 展示了 Job 中 Step 属性的 Schema 的定义，图 5-3 展示了配置文件中顶层 Step 属性的 Schema 的定义，Step 属性说明参见表 5-1。

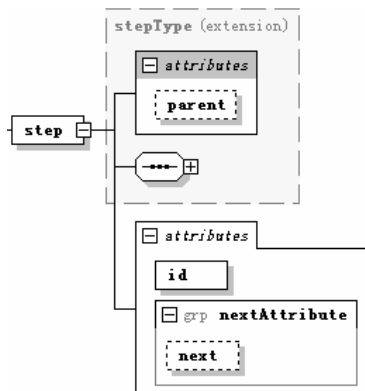


图 5-2 Step 属性 Schema 的定义

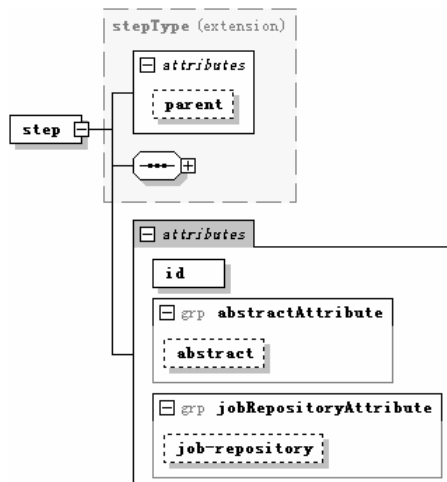


图 5-3 Step 顶层属性 Schema 的定义

表 5-1 Step 属性说明

属 性	说 明	默 认 值
id	Step 的唯一标识，在整个运行上下文中不允许重复	
next	当前 Step 执行完成后，next 元素指定下一个需要执行的 Step	
parent	定义当前 Step 的父 Step。Step 可以从其他 Step 继承。通常在父 Step 中定义共有的属性，在子 Step 中定义特有的属性。 如果父子都有的属性以子类中定义的为准，即子类属性会覆盖父类属性	
job-repository	定义该 Step 运行期间使用的 Step 仓库，默认使用名字为 jobRepository 的 Bean。 该属性只有在 Step 为顶层元素时候生效	jobRepository
abstract	定义当前 Step 是否是抽象的。True 表示当前 Step 是抽象的，不能被实例化。 该属性只有在 Step 为顶层元素时候生效	false

Step 主要子元素包括 tasklet（定义作业步的执行逻辑）、partition（任务分区）、job（引用独立 Job 作为作业步）、flow（引用 Flow 作为作业步）、next（下一个执行的作业步）、

stop（退出当前任务）、end（结束当前任务）、fail（使当前任务失败）、listeners（定义作业步拦截器）。图 5-4 展示了 Step 子元素的 Schema 的定义，Step 元素说明参见表 5-2。

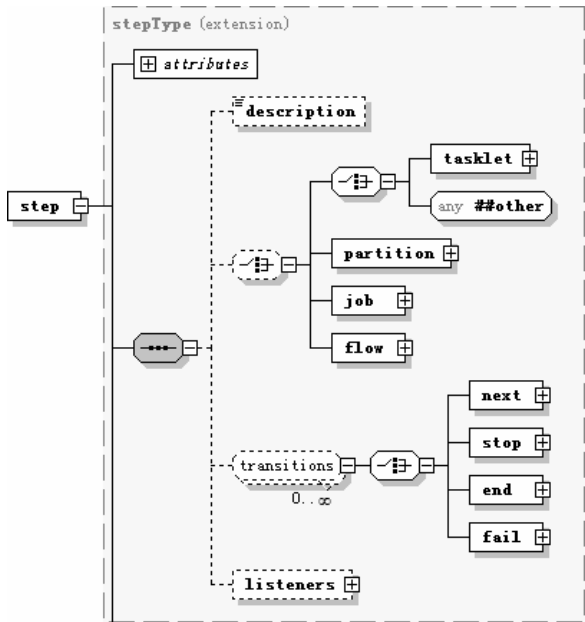


图 5-4 Step 子元素 Schema 的定义

表 5-2 Step 子元素说明

属 性	说 明
tasklet	定义具体作业步的执行逻辑
partition	定义当前任务是分区执行的，需要使用 partition 元素来声明 Step
job	引用独立配置的 Job 作为任务
flow	引用独立配置的 Flow 作为任务
next	根据退出状态定义下一步需要执行的 Step
stop	根据退出状态决定是否退出当前的任务，同时 Job 也会停止，作业状态为"STOPPED"
end	根据退出状态决定是否结束当前的任务，同时 Job 也会停止，作业状态为" COMPLETED"
fail	根据退出状态决定是否当前的任务失败，同时 Job 也会停止，作业状态为" FAILED"
listeners	定义作业步的拦截器

5.1.1 Step 抽象与继承

Spring Batch 框架支持抽象的 Step 定义和 Step 的继承特性。通过定义抽象的 Step 可以将 Step 的共性进行抽取，形成父类的 Step 定义；然后各个具体的 Step 可以继承父类 Step 的特

性，并定义自己的属性。通过 Step 的属性 `abstract` 可以定义抽象的 Step（抽象的 Step 不能被实例化执行），通过属性 `parent` 可以指定当前 Step 的父 Step。

抽象 Step

通过 `abstract` 属性可以指定 Step 为抽象的 Step。抽象的 Step 不能被实例化，只能作为其他 Step 的父。通常可以在抽象 Step 中定义全局性的配置，供子类使用。

代码清单 5-2 展示了典型的抽象 Step 的定义。

代码清单 5-2 抽象 Step 定义

```
1.     <step id="abstractParentStep" abstract="true">
2.         <tasklet>
3.             <chunk commit-interval="5" />
4.         </tasklet>
5.     </step>
```

其中，1 行：通过 `abstract` 属性，指定当前的 Step 为抽象的。

继承 Step

通过 `parent` 属性可以指定当前 Step 的父类，类似于 Java 世界中的继承一样，子类 Step 具有父类中定义的所有属性能力。子类继承时候，可以从抽象 Step 继承，也可以从普通的 Step 中继承。图 5-5 展示了 Step 间继承关系。

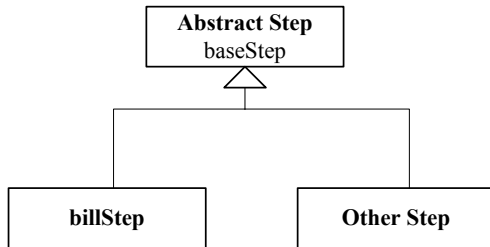


图 5-5 Step 继承

如果子类和父类都定义了相同的属性值，以子类中定义的为准。

代码清单 5-3 展示了典型的继承 Step 的定义。

代码清单 5-3 抽象 Step、继承 Step 配置

```
1.     <step id="parentStep" parent="abstractParentStep">
2.         <tasklet transaction-manager="transactionManager" allow-start-if-
3.             complete="true">
4.             <chunk reader="csvItemReader" writer="csvItemWriter" commit-
5.                 interval="10" />
6.         </tasklet>
7.     </step>
```

```

7.     <job id="billJob">
8.         <step id="billStep" parent="parentStep">
9.             <tasklet>
10.                <chunk processor="creditBillProcessor" commit-interval="2">
11.                    </chunk>
12.                </tasklet>
13.            </step>
14.        </job>

```

其中，1 行：定义 parentStep 的父为 abstractParentStep，继承抽象的 Step。

3 行：定义了 reader、writer、commit-interval 三个属性，这样子类 billStep 不用重复定义这三个属性，可直接使用父类 parentStep 中的定义。

10 行：子类 billStep 定义属性 processor、commit-interval 两个属性，因为属性 commit-interval 在父类和子类都有定义，因此使用子类 billStep 中定义的属性 commit-interval。

5.1.2 Step 执行拦截器

Spring Batch 框架在 Step 执行阶段提供了拦截器，使得在 Step 执行前后能够加入自定义的业务逻辑。Step 执行阶段拦截器接口：org.springframework.batch.core.StepExecutionListener。

StepExecutionListener 接口声明参见代码清单 5-4。

代码清单 5-4 StepExecutionListener 接口声明

```

1. public interface StepExecutionListener extends StepListener {
2.     void beforeStep(StepExecution stepExecution);
3.     ExitStatus afterStep(StepExecution stepExecution);
4. }

```

其中，2 行：表示在 Step 执行之前调用该方法。

3 行：表示在 Step 执行之后调用该方法，读者需要注意，该操作的返回值是 ExitStatus，表示当次作业步执行后的退出状态。

为 chunkStep 配置拦截器，参见代码清单 5-5。

代码清单 5-5 chunkStep 拦截器配置

```

1.     <job id="chunkJob">
2.         <step id="chunkStep">
3.             <tasklet>
4.                 <chunk reader="reader" writer="writer" commit-interval="10" />
5.             </tasklet>
6.             <listeners>
7.                 <listener ref="sysoutListener"></listener>
8.                 <listener ref="sysoutAnnotationListener"></listener>
9.             </listeners>
10.        </step>
11.    </job>

```

```

12.     <bean:bean id="sysoutAnnotationListener"
13.         class="com.juxtapose.example.ch05.listener.SystemOut" />

```

其中，7行：定义 step 的拦截器，sysoutListener 的实现参见示例项目代码 com.juxtapose.example.ch04.listener.SystemOutJobExecutionListener。

8行：定义 step 的拦截器，sysoutAnnotationListener 的实现参见示例项目代码 com.juxtapose.example.ch04.listener.SystemOut。

系统实现

Spring Batch 框架默认提供了 StepExecutionListener 的实现，分别完成不同功能，具体参见表 5-3 描述。

表 5-3 StepExecutionListener 默认实现

StepExecutionListener 默认实现	功能说明
CompositeStepExecutionListener	拦截器组合模式，支持一组拦截器调用
StepExecutionListenerSupport	StepExecutionListener 空实现，可以直接继承，仅复写关心的操作
StepListenerSupport	同时实现 StepExecutionListener、ChunkListener、ItemReadListener、ItemProcessListener、ItemWriteListener、SkipListener 接口的空实现，可以直接继承，仅复写关心的操作
MulticasterBatchListener	同时实现 StepExecutionListener、ChunkListener、ItemReadListener、ItemProcessListener、ItemWriteListener、SkipListener 接口的组合模式，支持一组拦截器调用

拦截器异常

拦截器方法如果抛出异常会影响 Step 的执行，所以在执行自定义的拦截器时候，需要考虑对拦截器发生的异常做处理，避免影响业务。

如果拦截器发生异常，会导致 Step 执行的状态为"FAILED"；作业的状态同样为"FAILED"。

执行顺序

在配置文件中可以配置多个 listener，拦截器之间的执行顺序按照 listener 定义的顺序执行。before 方法按照 listener 定义的顺序执行，after 方法按照相反的顺序执行。上面示例代码中执行的顺序如下：

- (1) sysoutListener 拦截器的 before 方法；
- (2) sysoutAnnotationListener 拦截器的 before 方法；
- (3) sysoutAnnotationListener 拦截器的 after 方法；
- (4) sysoutListener 拦截器的 after 方法。

Annotation

Spring Batch 框架提供了 Annotation 机制，可以不实现接口 `StepExecutionListener`，直接通过 Annotation 的机制定义拦截器。为 `StepExecutionListener` 提供的 Annotation 有：

- `@BeforeStep`
- `@AfterStep`

使用 Annotation 声明的拦截器的 Spring 配置文件和实现接口 `StepExecutionListener` 的拦截器配置一样，只需要在 `listeners` 节点中声明即可。

通过 Annotation 声明的拦截器代码，参见代码清单 5-6。

代码清单 5-6 Annotation 声明 Step 拦截器

```
1. public class Sysout {  
2.     @BeforeStep  
3.     public void beforeStep(StepExecution stepExecution) {  
4.         .....  
5.     }  
6.  
7.     @AfterStep  
8.     public ExitStatus afterStep(StepExecution stepExecution) {  
9.         .....  
10.    }  
11. }
```

merge 属性

假设有这样一个场景，所有的 Step 都希望拦截器 `sysoutListener` 能够执行，而拦截器 `sysoutAnnotationListener` 则由每个具体的 Step 定义是否执行，通过抽象和继承属性可以完成上面的场景。

merge 属性配置代码参见代码清单 5-7。

代码清单 5-7 merge 属性示例

```
1.     <step id="abstractParentStep" abstract="true">  
2.         <tasklet>.....</tasklet>  
3.         <listeners>  
4.             <listener ref="sysoutListener"></listener>  
5.         </listeners>  
6.     </step>  
7.     <job id="subChunkJob">  
8.         <step id="subChunkStep" parent="abstractParentStep">  
9.             <tasklet>.....</tasklet>  
10.            <listeners merge="true">  
11.                <listener ref="sysoutAnnotationListener"></listener>  
12.            </listeners>  
13.        </step>  
14.    </job>
```

其中,10行:通过 merge 属性,可以与父类中的拦截器配置进行合并,表示在 subChunkStep 中有两个拦截器会同时工作。

5.2 配置 Tasklet

tasklet 元素定义任务的具体执行逻辑,执行逻辑可以自定义实现,也可以使用 Spring Batch 的 Chunk 操作,提供了标准的读、处理、写三步操作。通过 tasklet 元素同样可以定义事务、处理线程、启动控制、回滚控制、拦截器等功能。

tasklet 元素的属性 Schema 定义,参见图 5-6。

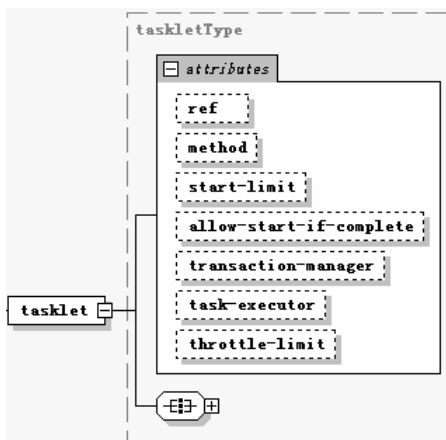


图 5-6 tasklet 属性 Schema 的定义

tasklet 属性说明,参见表 5-4。

表 5-4 tasklet 属性说明

属 性	说 明	默 认 值
ref	引用自定义实现 Tasklet 接口的 Bean 当使用自定义的 Tasklet 业务时候,必须使用 ref 来引用,如果使用 Spring Batch 提供的 Chunk 组件,请使用 chunk 元素来定义业务	
method	引用自定义的业务 Bean,需要调用的操作,对操作的要求: 参数需要和 Tasklet.execute 具有相同的参数。 返回值: 可以是 void、Boolean 或者 RepeatStatus	execute
start-limit	Step 能够启动的最大次数,超过最大次数后会抛出异常	Integer.MAX_VALUE
allow-start-if-complete	是否允许完成(状态为"COMPLETED")的 Step 重新启动	false
transaction-manager	tasklet 配置的事务管理器,控制业务的处理操作	

续表

属 性	说 明	默 认 值
task-executor	任务执行处理器，定义后表示采用多线程执行任务，需要考虑多线程执行任务时候的安全性	
throttle-limit	最大使用线程池的数目	6

tasklet 元素的子元素 Schema 定义，参见图 5-7。

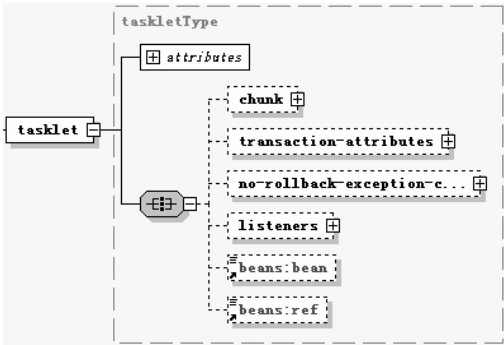


图 5-7 tasklet 子元素 Schema 的定义

tasklet 子元素说明，参见表 5-5。

表 5-5 tasklet 子元素说明

属 性	说 明	默 认 值
chunk	定义面向批的业务操作，使用 chunk 可以复用 Spring Batch 提供的基础设施	
transaction-attributes	定义任务具体的事务属性，例如隔离级别、事务传播方式、超时时间等	
no-rollback-exception-classes	定义不会触发事务回滚的异常。 通常情况下，作业处理过程中发生的任何异常都会导致作业事务的回顾，Spring Batch 框架提供指定异常不影响事务的能力	
listeners	定义具体任务级别的拦截器	

5.2.1 重启 Step

批处理作业框架需要支持任务重新启动，批处理作业处理数据发生错误时，在数据修复后需要能够将当前的任务实例执行完毕。在基本概念章节我们已经介绍了 Job Instance、JobParameters、ExecutionContext 之间的关系。

Spring Batch 框架支持状态为非"COMPELETED"的 Job 实例重新启动，Job 实例重启的时候，会从当前失败的 Step 重新开始执行，同时可以通过 start-limit 属性控制任务启动的次数。

启动次数限制

默认情况下，作业实例可以无限次地重复启动。在有些场景下需要限制作业实例的启动次数，例如在执行任务分区（particular）的 Step 中，需要对分区的 Step 限制启动次数为 1 次，因为在数据错误的情况下，多次重启 Step 没有任何意义。当然，读者可以找到更多的场景来使用限制任务重启的次数。代码清单 5-8 展示了作业步 startLimitStep 只能启动一次。

代码清单 5-8 配置 Step 启动次数

```
1. <step id="startLimitStep">
2.     <tasklet start-limit="1">
3.         <chunk reader="reader" processor="processor" writer="writer"/>
4.     </tasklet>
5. </step>
```

定义 startLimitStep 仅能启动一次，第二次启动的时候会抛出异常。start-limit 默认值是 Integer.MAX_VALUE，表示任务可以无限次地启动。

重启已完成的任務

默认情况下，Job Instance 重新启动的时候，已经完成的任務不会再次被执行。仅在某些特殊场景下，已经完成的 Step 在任务重启的时候需要再次执行，这可以通过属性 allow-start-if-complete 来设置。代码清单 5-9 展示了作业步重启的配置。

代码清单 5-9 配置 Step 支持重启

```
1. <step id="allowStartStep">
2.     <tasklet allow-start-if-complete="true">
3.         <chunk reader="reader" processor="processor" writer="writer"/>
4.     </tasklet>
5. </step>
```

其中，2 行：通过设置属性 allow-start-if-complete 值为 true，表示已经完成的 Step 可以被再次启动。

5.2.2 事务

Spring Batch 框架提供了事务能力保障 Job 可靠地执行，能够将 Job 的 read、process 和 write 三者有效地控制在一起，保证操作的完整性；Job 执行期间的元数据状态的持久化同样依赖事务的保证。在 Job 执行期间，可以配置事务管理器、事务的基本属性（包括隔离级别、传播方式、事务超时等信息）。

事务管理器

为了使用事务管理器，需要在配置文件中声明标准 Spring 方式的事务管理器，参见代码清单 5-10 中配置的事务管理器。

代码清单 5-10 配置事务管理器

```
1. <bean:bean id="transactionManager"
2.     class="org.springframework.jdbc.datasource.
3.         DataSourceTransactionManager">
4.     <bean:property name="dataSource" ref="dataSource" />
5. </bean:bean>
```

其中，2 行：指定使用的事务管理器类。

3 行：指定使用的数据源。

事务管理器需要提供指定的数据源对象，在对此数据源的任何操作将会得到事务的保障。事务管理器确定以后，需要为 tasklet 定义使用的事务管理器，参见代码清单 5-11。

代码清单 5-11 为 tasklet 配置事务管理器

```
1. <step id="chunkStep">
2.     <tasklet transaction-manager="transactionManager" start-limit="1">
3.         <chunk reader="reader" processor="processor" writer="writer"
4.             commit-interval="5" />
5.     </tasklet>
6. </step>
```

示例配置中使用 transactionManager 对象，保障 chunkStep 作业的操作在一个事务内完成。

事务属性

在事务管理器中可以指定事务的属性，例如事务的隔离级别、事务的传播方式、事务超时时间等，这些事务属性同样可以在 tasklet 中配置，详细定义任务的事务属性定义。

事务的隔离级别（**isolation**）参见表 5-6 描述。

表 5-6 事务的隔离级别（**isolation**）

隔离级别	说 明
SERIALIZABLE	最严格的级别，事务串行执行，资源消耗最大
REPEATABLE_READ	保证了一个事务不会修改已经由另一个事务读取但未提交（回滚）的数据。避免了“脏读取”和“不可重复读取”的情况，但是带来了更多的性能损失
READ_COMMITTED	默认事务等级，保证了一个事务不会读取另一个并行事务已修改但未提交的数据，避免了“脏读取”。该级别适用于大多数系统
READ_UNCOMMITTED	保证了读取过程中不会读取非法数据

事务传播方式（**propagation**）参见表 5-7 描述。

表 5-7 事务传播方式（**propagation**）

传播方式	说 明
REQUIRED	支持当前事务，如果当前没有事务，就新建一个事务
SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行

续表

传播方式	说 明
MANDATORY	支持当前事务，如果当前没有事务，就抛出异常
REQUIRES_NEW	新建事务，如果当前存在事务，则把当前事务挂起
NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
NEVER	以非事务方式执行，如果当前存在事务，则抛出异常

代码清单 5-12 给出了如何配置事务的基本属性。

代码清单 5-12 配置事务基本属性

```

1. <step id="chunkStep">
2.     <tasklet transaction-manager="transactionManager" start-limit="1">
3.         <chunk reader="reader" processor="processor" writer="writer"
4.             commit-interval="5" />
5.         <transaction-attributes isolation="DEFAULT"
6.             propagation="REQUIRED" timeout="30"/>
7.     </tasklet>
8. </step>

```

其中，4 行：定义事务的隔离级别为 DEFAULT，事务传播方式为 REQUIRED（表示需要事务，如果外面有事务则加入外部事务，如果外部没有事务则新启一个事务），事务超时时间为 30 秒。

5.2.3 事务回滚

通过事务控制可以较好地保证事务任务的执行。在业务处理过程中，包括读、写、处理数据如果发生了异常会导致事务回滚，Spring Batch 框架提供了发生特定异常不触发事务回滚的能力，可以在 tasklet 中通过子元素 no-rollback-exception-classes 来定义特定异常。

配置不触发回滚操作异常的配置，参见代码清单 5-13。

代码清单 5-13 配置不触发回滚操作异常

```

1. <step id="chunkStep">
2.     <tasklet transaction-manager="transactionManager" start-limit="1">
3.         <chunk reader="reader" processor="processor" writer="writer"
4.             commit-interval="5" />
5.         <transaction-attributes isolation="DEFAULT"
6.             propagation="REQUIRED" timeout="30"/>
7.         <no-rollback-exception-classes>
8.             <include class="org.springframework.batch.item.validator.
9.                 ValidationException"/>
10.        </no-rollback-exception-classes>
11.    </tasklet>
12. </step>

```

其中，5~7 行：配置发生 `ValidationException` 异常或者其子类异常时，不会触发事务的回滚操作；其他任何类型异常都会导致事务的回滚操作。

5.2.4 多线程 Step

Job 执行默认情况使用单个线程完成任务的执行。Spring Batch 框架支持为 Step 配置多个线程，即可以使用多个线程并行执行一个 Step，可以提高 Step 的处理速度。使用 `tasklet` 的属性 `task-executor` 为 Step 定义多线程。示例代码参见代码清单 5-14。

代码清单 5-14 配置多线程 Step

```
1. <job id="multiThreadJob">
2.   <step id="multiThreadStep">
3.     <tasklet task-executor="taskExecutor" throttle-limit="6">
4.       <chunk reader="reader" processor="processor" writer="writer"
5.         commit-interval="5"/>
6.     </tasklet>
7.   </step>
8. </job>
9. <bean:bean id="taskExecutor"
10.   class="org.springframework.scheduling.concurrent.
11.     ThreadPoolTaskExecutor">
12.   <bean:property name="corePoolSize" value="5"/>
13.   <bean:property name="maxPoolSize" value="15"/>
14. </bean:bean>
```

其中，3 行：通过属性 `task-executor` 定义执行 Step 需要的线程池，属性 `throttle-limit` 用于限制能使用的最大的线程数目。

9~13 行：定义线程池，默认有 5 个线程，最大为 15 个线程。

5.2.5 自定义 Tasklet

元素 `tasklet` 中可以配置面向 `Chunk` 的任务，或者是开发者实现自定义的 `Tasklet`。两种都可以完成业务逻辑的配置。面向 `Chunk` 的操作在下一节详细描述。自定义的 `Tasklet` 需要接口 `org.springframework.batch.core.step.tasklet.Tasklet` 来实现。

接口 `Tasklet` 声明，参见代码清单 5-15。

代码清单 5-15 接口 `Tasklet` 声明

```
1. public interface Tasklet {
2.   RepeatStatus execute(StepContribution contribution, ChunkContext
3.     chunkContext) throws Exception;
4. }
```

`execute()`是接口 `Tasklet` 唯一必须实现的方法，该方法完成对业务的处理。通过自定义的 `Tasklet` 可以复用开发者已有的企业服务。

可以通过 `tasklet` 的属性 `ref` 和属性 `method` 来使用自定义的 `Tasklet`。配置示例参见代码清单 5-16。

代码清单 5-16 配置 `tasklet` 的 `ref`

```
1.      <job id="custTaskletJob">
2.          <step id="custTaskletStep">
3.              <tasklet ref="helloWorldTasklet">
4.                  </tasklet>
5.              </step>
6.          </job>
7. <bean:bean id="helloWorldTasklet" class="com.juxtapose.example.ch05.
    HelloWorldTasklet" />
```

其中，3 行：通过 `ref` 属性引用自定义的业务操作。
7 行：定义自己实现 `Tasklet` 接口的业务 `HelloWorldTasklet`。

Tasklet 系统实现

Spring Batch 框架默认提供了 `Tasklet` 的实现，分别完成不同功能，在实际应用开发过程中可以直接使用系统默认的实现，也可以重新实现自己的 `Tasklet`。

系统提供的默认 `Tasklet` 实现，参见表 5-8。

表 5-8 Tasklet 默认实现

Tasklet 默认实现	功能说明
CallableTaskletAdapter	Callable 接口适配器 org.springframework.batch.core.step.tasklet.callableTaskletAdapter
ChunkOrientedTasklet	面向批的任务处理，用于 Chunk 的处理操作 org.springframework.batch.core.step.item.ChunkOrientedTasklet<I>
MethodInvokingTaskletAdapter	用于适配已有服务，通过代理的方式调用已经存在的服务 org.springframework.batch.core.step.tasklet.MethodInvokingTaskletAdapter
SystemCommandTasklet	系统命令任务类，可以调用执行的命令 org.springframework.batch.core.step.tasklet.SystemCommandTasklet

接下来我们以 `MethodInvokingTaskletAdapter` 为例子，展示如何使用系统默认提供的 `Tasklet`。框架提供的其他默认 `Tasklet`，读者可以自行深入研究。

使用 `MethodInvokingTaskletAdapter`，通过代理的方式调用已经存在的服务，代码清单 5-17 展示了在 `Job` 中调用查询 `jobRegistry` 对象的所有作业名的操作。

代码清单 5-17 配置代理服务

```

1.     <job id="taskletAdapterJob">
2.         <step id="taskletAdapterStep">
3.             <tasklet ref="adapter">
4.                 </tasklet>
5.             </step>
6.         </job>
7.
8.         <bean:bean id="adapter"
9.             class="org.springframework.batch.core.step.tasklet.
                MethodInvokingTaskletAdapter">
10.            <bean:property name="targetObject" ref="jobRegistry" />
11.            <bean:property name="targetMethod" value="getJobNames" />
12.        </bean:bean>

```

其中，3 行：自定义的 Tasklet 使用 MethodInvokingTaskletAdapter 对象。

8~12 行：定义 adapter，通过 MethodInvokingTaskletAdapter 调用对象 jobRegistry 的 getJobNames 操作。

说明：引用自定义的业务 Bean，需要调用的操作，对操作有如下的要求：

参数需要和 Tasklet.execute 具有相同的参数；

返回值可以是 void、Boolean 或者 RepeatStatus。

5.3 配置 Chunk

Chunk 元素定义面向批的处理操作，Chunk 典型地提供了标准的 read、process、write 三种操作，同时 Spring Batch 框架提供了丰富的读、写的组件，包括对格式化文件的读/写、xml 文件的读/写、数据库的读/写、JMS 消息的读/写等技术组件，可以在配置文件中直接使用。

Chunk 元素提供了功能性以外的能力，包括异常处理、批处理的可靠性、稳定性、异常重入的能力。具体包括事务提交间隔、跳过策略、重试策略、读事务队列、批处理完成策略等。

Chunk 元素的属性 Schema 定义参见图 5-8。

Chunk 属性说明，参见表 5-9。

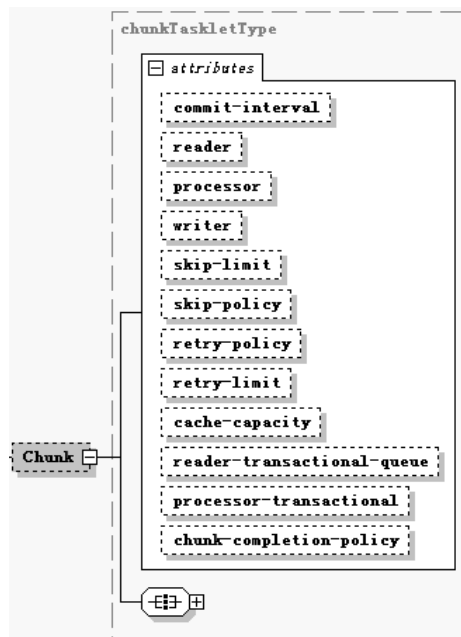


图 5-8 Chunk 属性 Schema 定义

表 5-9 Chunk 属性说明

属 性	说 明	默 认 值
commit-interval	提交间隔，读出、处理指定的数据后，通过 writer 批量写入，并提交事务	
reader	读取数据的 Bean 的名字，需要实现接口： <code>org.springframework.batch.item.ItemReader<T></code>	
processor	批处理中的处理逻辑，需要实现接口： <code>org.springframework.batch.item.ItemProcessor<I, O></code>	
writer	写数据的 Bean 的名字，需要实现接口： <code>org.springframework.batch.item.ItemWriter<T></code>	
skip-limit	异常发生时，允许跳过的最大数。 当跳过数目达到后，数据处理（包括读数据、处理数据、写数据）发生的异常会抛出，导致任务 Step 失败	
skip-policy	跳过策略 Bean，需要实现接口： <code>org.springframework.batch.core.step.skip.SkipPolicy</code>	
retry-policy	重试策略 Bean，需要实现接口： <code>org.springframework.batch.retry.RetryPolicy</code>	
retry-limit	任务执行重试的最大次数	
cache-capacity	retry-policy 缓存的大小，缓存用于存放重试上下文 <code>RetryContext</code> ，如果超过配置最大值，会发生异常： <code>org.springframework.batch.retry.policy.RetryCacheCapacityExceededException</code>	4096
reader-transactional-queue	是否从一个事务性的队列读取数据： <code>true</code> 表示从一个事务性的队列中读取数据，一旦发生异常会导致事务回滚，从队列中读取的数据同样会被重新放回到队列中； <code>false</code> 表示从一个没有事务的队列获取数据，一旦发生异常会导致事务回滚，消费掉的数据不会重新放置在队列中	false
processor-transactional	处理数据是否在事务中， <code>true</code> 表示将 processor 处理的结果放在缓存中，当执行重试或者跳过策略时可以看到缓存中处理的数据； <code>false</code> 表示不会将 processor 处理的数据放在缓存中，即 processor 在 chunk 的每一条记录仅会执行一次。 需要注意：如果将 reader-transactional-queue 设置为 <code>true</code> ，则 processor-transactional 必须设置为 <code>true</code>	true
chunk-completion-policy	批处理完成策略，需要实现接口： <code>org.springframework.batch.repeat.CompletionPolicy</code>	

Chunk 的子元素 Schema 定义，参见图 5-9。

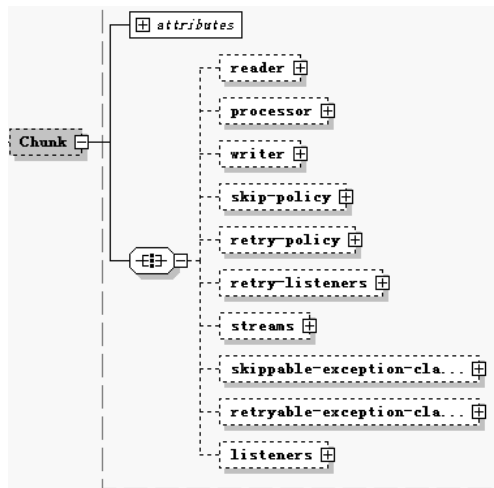


图 5-9 Chunk 子元素 Schema 定义

Chunk 子元素说明，参见表 5-10。

表 5-10 Chunk 子元素说明

属 性	说 明	默 认 值
reader	读取数据的 Bean 的定义，需要实现接口： <code>org.springframework.batch.item.ItemReader<T></code>	
processor	批处理中的处理逻辑的定义，需要实现接口： <code>org.springframework.batch.item.ItemProcessor<I, O></code>	
writer	写数据的 Bean 的定义，需要实现接口： <code>org.springframework.batch.item.ItemWriter<T></code>	
skip-policy	跳过策略 Bean 定义，需要实现接口： <code>org.springframework.batch.core.step.skip.SkipPolicy</code>	
retry-policy	重试策略 Bean 定义，需要实现接口： <code>org.springframework.batch.retry.RetryPolicy</code>	
retry-listeners	重试操作监听器，需要实现接口： <code>org.springframework.batch.retry.RetryListener</code>	
streams	定义一组实现 <code>ItemStream</code> 的对象，需要实现接口： <code>org.springframework.batch.item.ItemStream</code> 。 Step 执行期间需要知道 reader、processor、writer 定义的实例哪些是实现接口 <code>ItemStream</code> 的（显现接口 <code>ItemStream</code> 的 reader、processor、writer 的对象能够在任务重启的时候从正确的点恢复）。Spring Batch 框架自动注册实现了接口 <code>ItemStream</code> 的对象，如果在 reader、processor、writer 中用到的对象没有直接实现接口 <code>ItemStream</code> ，需要在此处显式注册	

续表

属 性	说 明	默 认 值
skippable-exception-classes	定义一组触发跳过的异常	
retryable-exception-classes	定义一组触发重试的异常	
listeners	定义一组 Chunk 的拦截器，需要实现接口： org.springframework.batch.core.ChunkListener	

5.3.1 提交间隔

批处理作业通常针对大数据量进行处理，同时框架需要将作业处理的状态实时地持久化到数据库中，如果读取一条记录就进行写操作或者状态数据的提交，会大量消耗系统资源，导致批处理框架性能度差。在面向批处理 Chunk 的操作中，可以通过属性 `commit-interval` 设置 read 多少条记录后进行一次提交。通过设置 `commit-interval` 的间隔值，减少提交频次，降低资源使用率。

面向 Chunk 的操作序列图，参见图 5-10。

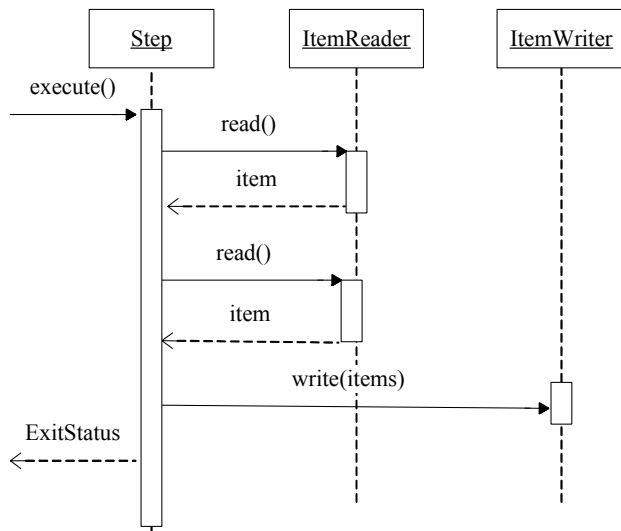


图 5-10 面向 Chunk 的操作序列图

通过 `commit-interval` 设置 Chunk 的提交频次，示例代码参见代码清单 5-18。

代码清单 5-18 设置 chunk 提交频次

```

1.     <job id="chunkJob">
2.         <step id="chunkStep">
3.             <tasklet transaction-manager="transactionManager" start-limit="1">
4.                 <chunk reader="reader" processor="processor" writer="writer"

```

```

5.         commit-interval="5"/>
6.     </tasklet>
7. </step>
8. </job>

```

其中，5 行：设置提交间隔属性 `commit-interval` 为 5，表示每读取 5 条记录后，进行一次写操作，同时将执行的状态数据通过 `JobRepository` 持久化到数据库中。可以将提交间隔设置的值作为作业中处理的最基本的单元，通过事务管理器保证了操作的一致性。

按照面向 `Chunk` 的操作，如果提交间隔是 5 次，那么读操作被调用 5 次，写操作被调用 1 次。读 `Item` 被汇总到列表中，最终被统一写出。读/写操作被封装在同一个事务块中，当前批次的作业步执行完毕后框架通过 `JobRepository` 将状态数据保存。

使用伪代码展示执行动作，参见代码清单 5-19。

代码清单 5-19 伪代码展示提交间隔

```

1. TransactionManager tm = ...;
2. JobRepository jobRepository=...;
3. try{
4.     tm.begin();
5.     List items = new ArrayList();
6.     for(int i = 0; i < commitInterval; i++){
7.         items.add(itemReader.read());
8.     }
9.     itemWriter.write(items);
10.    tm.commit;
11. }catch(Throwable t){
12.    tm.rollback();
13. }finally{
14.    jobRepository.addOrUpdate(batchMetaData); //状态数据保存
15. }

```

其中，4 行：事务开始。

5~9 行：根据 `commit-interval` 设置的值 5，读取 5 条数据，然后执行写。

10 行：事务提交。

12 行：发生异常回滚事务。

14 行：无论执行成功还是失败，都要将 `Chunk` 执行的状态数据持久化。

5.3.2 异常跳过

设想前面的信用卡对账单的处理的业务场景，银行每天需要处理海量的对账文件，如果对账文件中有少量的一行或者几行错误格式的记录，在真正进行作业处理的时候，不希望因为几行错误的记录而导致整个作业的失败；而是希望将这几行没有处理的记录跳过去，让整个 `Job` 正确执行，对于错误的记录则通过日志的方式记录下来后续进行单独的处理。

对账单格式错误文件，参见代码清单 5-20。

代码清单 5-20 格式错误对账单

```
1. 4047390012345678,tom,100.00,xxx-yyy-zzz,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.20,2013-2-9 15:15:37
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road
```

其中，1 行：交易日期格式有误。

4 行：记录不完整，缺少交易的地址列信息。

如果记录行数错误较多，即使成功执行完毕 Job，也没有任何意义，因为错误记录数太多导致后续的手工修复可能更复杂。此时最好的处理方式是让 Job 执行失败，然后修复记录文件重新执行 Job 作业。Spring Batch 框架提供了最大跳过记录数的限制，当跳过的记录数大于设定的值后，Job 作业将会失败。

Spring Batch 框架通过属性 skip-limit、skippable-exception-classes、skip-policy 来完成跳过能力。

skippable-exception-classes：定义可以对记录跳过的异常，还可以定义跳过一组异常，如果发生了定义的异常或者子类异常都不会导致作业失败。

skip-limit：任务处理发生异常时，允许跳过的最大次数。

skip-policy：当默认的按照次数跳过策略不能满足需求时，可以配置自定义跳过策略，需要实现接口 org.springframework.batch.core.step.skip.SkipPolicy。

配置 Skip

配置 Skip 的配置文件参见代码清单 5-21。

代码清单 5-21 配置作业 Skip

```
1. <job id="skipJob">
2.   <step id="skipStep">
3.     <tasklet>
4.       <chunk reader="reader" processor="processor" writer="writer"
5.         commit-interval="1" skip-limit="20">
6.         <skippable-exception-classes>
7.           <include class="java.lang.RuntimeException" />
8.           <exclude class="java.io.FileNotFoundException" />
9.         </skippable-exception-classes>
10.      </chunk>
11.    </tasklet>
12.  </step>
13. </job>
14.
```

```

15.     <bean:bean id="processor"
16.         class="com.juxtapose.example.ch05.RadomExceptionItemProcessor" />

```

其中，5 行：异常发生时，允许跳过最大的异常次数为 20 次，如果超过 20 次后，再次发生的异常会导致 Job 的失败。

6~9 行：include 定义支持跳过的异常，exclude 定义排出在外的异常；当发生任何 RuntimeException 类型异常（同时 FileNotFoundException 排出在外）时会跳过处理，但是当已经跳过 20 次后，第 21 次发生异常的时候，会导致 Job 的失败，因为超过了 skip-limit="20" 的限制。

15~16 行：自定义一个 ItemProcessor 实现类 RadomExceptionItemProcessor，该类会随机发生 RuntimeException 类型异常。

com.juxtapose.example.ch05.RadomExceptionItemProcessor 实现代码，参见代码清单 5-22。

代码清单 5-22 RadomExceptionItemProcessor 实现类

```

1.  public class RadomExceptionItemProcessor implements ItemProcessor<String,
    String> {
2.      Random ra = new Random();
3.
4.      public String process(String item) throws Exception {
5.          int i = ra.nextInt(10);
6.          System.out.println("Process " + item + "; Random i=" + i);
7.          if(i%2 == 0){
8.              throw new RuntimeException("make error!");
9.          }else{
10.             return item;
11.          }
12.      }
13. }

```

其中，5 行：定义随机数。

7~10 行：根据对 2 取模，抛出异常或者正确返回记录。

异常跳过的完整使用请参见第 10 章健壮 Job，包括异常策略配置，异常拦截器的使用。

5.3.3 Step 重试

Step 执行期间 read、process、write 发生的任何异常都会导致 Step 执行失败，进而导致作业的失败。批处理作业的自动化、定时触发，有特定的执行时间窗口特性，决定了尽可能地减少 Job 的失败。处理任务阶段发生的异常可以让业务失败，也可以通过 Skip 的设置，跳过部分异常；但是另外还有部分异常，例如并发对数据库的操作导致的数据库锁的异常（DeadlockLoserDataAccessException）和网络不稳定导致的网络连接异常（java.net.ConnectException）。这类异常的出现可能在下次重新操作的时候消失，数据库锁的异常在下次操作可能正确恢复，网络不能连接的异常可能在重试几次后恢复正常。因此，这些异常出现的时

候，不期望作业发生异常，也不期望跳过处理，而是希望通过几次重试操作，尽可能地让 Job 成功执行。

Spring Batch 框架提供了任务重试功能，重试次数限制功能、自定义重试策略以及重试拦截器能力。分别通过 `retryable-exception-classes`、`retry-limit`、`retry-policy`、`cache-capacity`、`retry-listeners` 来实现。

retryable-exception-classes：定义可以重试的异常，可以定义一组重试的异常，如果发生了定义的异常或者子类异常都会导致重试。

retry-limit：任务执行重试的最大次数。

retry-policy：定义自定义的重试策略，需要实现接口 `org.springframework.batch.retry.RetryPolicy`。

cache-capacity：`retry-policy` 缓存的大小，缓存用于存放重试上下文 `RetryContext`，如果超过配置最大值，会发生异常 `org.springframework.batch.retry.policy.RetryCacheCapacityExceededException`。

retry-listeners：配置重试监听器，监听器需要实现接口 `org.springframework.batch.retry.RetryListener`。

配置 Retry

配置 Retry 的配置文件，参见代码清单 5-23。

代码清单 5-23 配置 Retry

```
1.     <job id="retryJob">
2.         <step id="retryStep">
3.             <tasklet>
4.                 <chunk reader="reader" processor="alwaysExceptionItemProcessor"
5.                     writer="writer" commit-interval="1" retry-limit="3">
6.                     <retry-listeners>
7.                         <listener ref="sysoutRetryListener"></listener>
8.                     </retry-listeners>
9.                     <retryable-exception-classes>
10.                        <include class="java.lang.RuntimeException" />
11.                        <exclude class="java.io.FileNotFoundException" />
12.                    </retryable-exception-classes>
13.                </chunk>
14.            </tasklet>
15.        </step>
16.    </job>
17.    <bean:bean id="sysoutRetryListener"
18.        class=" com.juxtapose.example.ch05.listener.SystemOutRetry
19.            Listener" />
20.    <bean:bean id="alwaysExceptionItemProcessor"
```

```
20.         class="com.juxtapose.example.ch05.AlwaysExceptionItem  
Processor" />
```

其中，5 行：属性 `retry-limit` 定义最大重试次数为 3 次，当重试超过 3 次后发生的异常会导致 Step 失败。

6~8 行：定义重试操作的拦截器，在 `Chunk` 发生重试操作时候，会触发拦截器 `sysoutRetryListener` 操作。

9~12 行：属性 `retryable-exception-classes` 定义重试的异常，可以定义多个重试的异常，`include` 表示能够触发重试的异常，`exclude` 表示不会触发重试的异常。

17~18 行：定义重试拦截器 `sysoutRetryListener`，拦截器仅把信息打印在控制台上。

19~20 行：定义自定义处理器 `alwaysExceptionItemProcessor`，每次操作均会发生异常，用于模拟 `Chunk` 发生异常。

`com.juxtapose.example.ch05.AlwaysExceptionItemProcessor` 实现，参见代码清单 5-24。

代码清单 5-24 `AlwaysExceptionItemProcessor` 实现类

```
1.  public class AlwaysExceptionItemProcessor implements ItemProcessor<String,  
String> {  
2.      Random ra = new Random();  
3.      public String process(String item) throws Exception {  
4.          int i = ra.nextInt(10);  
5.          if(i%2 == 0){  
6.              System.out.println("Process " + item + "; Random i=" + i + "; ...");  
7.              throw new MockARuntimeException("make error!");  
8.          }else{  
9.              System.out.println("Process " + item + "; Random i=" + i + "; ...");  
10.             throw new MockBRuntimeException("make error!");  
11.         }  
12.     }  
13. }
```

其中，4 行：定义随机数。

5~10 行：根据对 2 取模，抛出不同类型的异常。

5.3.4 `Chunk` 完成策略

面向 `Chunk` 的操作执行期间，根据设置的提交间隔 `commit-interval` 值，当读数据达到提交间隔后，执行一次提交操作，然后重复执行 `Chunk` 的读操作，直到再次达到间隔值。`Spring Batch` 框架除了提供 `commit-interval` 能力外，该框架还提供了 `Chunk` 完成策略能力，通过完成策略可以配置任务的提交时机，`Chunk` 完成策略的定义接口为 `org.springframework.batch.repeat.CompletionPolicy`。通过属性 `chunk-completion-policy` 定义批处理的完成策略。

面向 `Chunk` 的完成策略顺序参见图 5-11。

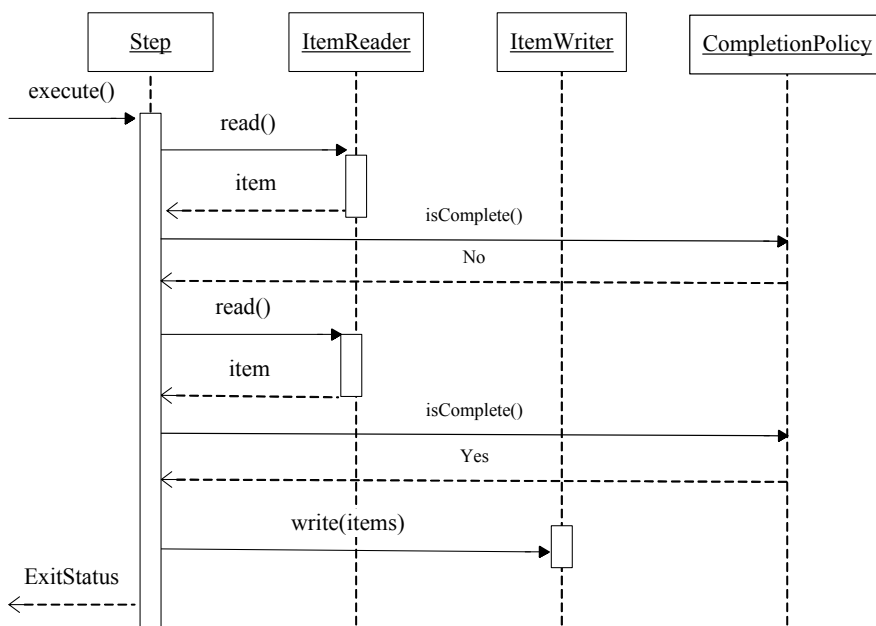


图 5-11 面向 Chunk 的完成策略顺序图

说明：chunk-completion-policy 定义批处理完成策略，不是表示任务的完成策略，Chunk 执行期间是按照 Chunk 完成策略执行批量提交的，批量提交会执行一次写操作，同时将批处理的状态数据通过 JobRepository 持久化。

说明：属性 chunk-completion-policy 和属性 commit-interval 不能同时存在；在 Chunk 中至少定义这两个其中的一个。即要么定义属性 chunk-completion-policy，要么定义属性 commit-interval。

CompletionPolicy 接口定义，参见代码清单 5-25。

代码清单 5-25 CompletionPolicy 接口定义

```

1. public interface CompletionPolicy {
2.     boolean isComplete(RepeatContext context, RepeatStatus result);
3.     boolean isComplete(RepeatContext context);
4.     RepeatContext start(RepeatContext parent);
5.     void update(RepeatContext context);
6. }
  
```

isComplete()定义当前重复操作是否完成，start()表示批处理操作开始，update()完成更新当前批处理的状态数据。

系统提供的默认 CompletionPolicy 实现列表参见表 5-11。

接下来以 SimpleCompletionPolicy 为例展示如何使用 CompletionPolicy。读者可以自行研究其他完成策略的使用。

表 5-11 完成策略 CompletionPolicy 默认实现

CompletionPolicy 默认实现	功能说明
CompletionPolicySupport	完成策略实现类，可以基于此实现类继承实现自定义的完成策略。通常在 Spring Batch 中以 Support 为结尾的实现类，仅仅提供了接口的默认实现，方便继承类，仅覆写感兴趣的方法，不用实现接口中的所有方法 org.springframework.batch.repeat.policy.CompletionPolicySupport
DefaultResultCompletionPolicy	完成策略默认实现类，根据 RepeatStatus 是否为 null 或者 RepeatStatus 允许继续判断是否完成 org.springframework.batch.repeat.policy.DefaultResultCompletionPolicy
SimpleCompletionPolicy	根据 chunkSize 值的大小，决定是否完成。最终 commit-intervval 是通过 SimpleCompletionPolicy 来实现的 org.springframework.batch.repeat.policy.SimpleCompletionPolicy
TimeoutTerminationPolicy	根据 Chunk 执行的时间来判断是否完成，在给定的时间内 chunk 会重复执行，直到超过给定的时间 org.springframework.batch.repeat.policy.TimeoutTerminationPolicy
CompositeCompletionPolicy	组合完成策略，可以将多个完成策略组合在一起使用，按照顺序判断是否应该完成；多个组合策略中只要有一个满足完成条件，表示完成则组合完成策略 org.springframework.batch.repeat.policy.CompositeCompletionPolicy

配置 SimpleCompletionPolicy，参见代码清单 5-26。

代码清单 5-26 配置简单完成策略 SimpleCompletionPolicy

```

1.     <job id="completionPolicyJob">
2.         <step id="completionPolicyStep">
3.             <tasklet>
4.                 <chunk reader="reader" processor="processor" writer="writer"
5.                     chunk-completion-policy="completionPolicy">
6.                 </chunk>
7.             </tasklet>
8.         </step>
9.     </job>
10.    <bean:bean id="completionPolicy"
11.        class="org.springframework.batch.repeat.policy.
12.            SimpleCompletionPolicy" >
13.        <bean:property name="chunkSize" value="5" />
14.    </bean:bean>
15.    <bean:bean id="processor"
16.        class=" org.springframework.batch.item.support.
17.            PassThroughItemProcessor" />

```

其中，5 行：利用属性 chunk-completion-policy 定义 Chunk 的完成策略为 completionPolicy，completionPolicy 默认使用 SimpleCompletionPolicy。

10~12 行：定义 SimpleCompletionPolicy，配置属性 chunkSize 为 5，表示每处理 5 次数据后进行一次事务提交；然后重复执行 chunk 操作。

14~15 行：定义 chunk 的处理器为 PassThroughItemProcessor，该处理器不做任何业务处理，直接将 read 读取的数据返回。

5.3.5 读、处理事务

读事务队列

reader-transactional-queue：是否从一个事务性的队列读取数据，当 reader 从 JMS 的消息队列获取数据时候，此属性生效，默认值为 false。

true 表示从一个事务性的队列中读取数据，一旦发生异常会导致事务回滚，从队列中读取的数据同样会被重新放回到队列中；false 表示从一个没有事务的队列获取数据，一旦发生异常导致事务回滚，消费掉的数据不会重新放置在队列中。

读 JMS 消息的 ItemReader 的具体使用，请参见 6.6 章节。

处理事务

processor-transactional：处理数据是否在事务中，true 表示在一次 Chunk 处理期间将 processor 处理的结果放在缓存中，当执行重试或者跳过策略时可以看到缓存中处理的数据，在写操作完成前可以重新执行 processor；false 表示在一次 Chunk 处理期间不会将 processor 处理的数据放在缓存中，即 processor 在 chunk 执行期间每一条记录仅会执行一次。

事务性 Processor 处理过程图参见图 5-12。

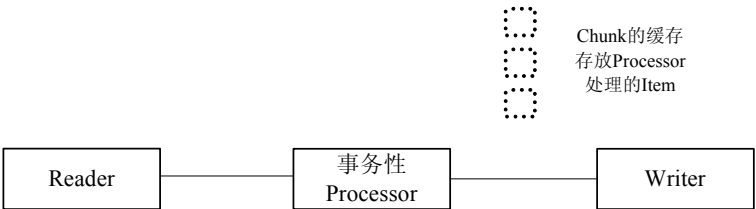


图 5-12 事务性 Processor 处理过程图

非事务性 Processor 处理过程图参见图 5-13。

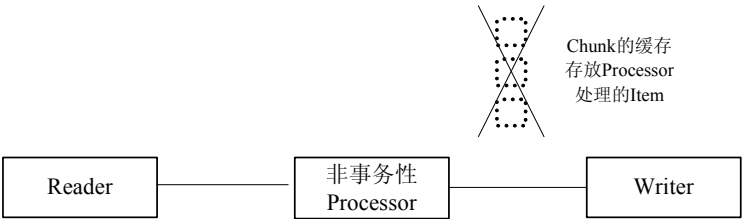


图 5-13 非事务性 Processor 处理过程图

说明：如果将 `reader-transactional-queue` 设置为 `true`，则 `processor-transactional` 必须设置为 `true`。

下面的示例，参见代码清单 5-27，它说明属性 `processor-transactional` 的具体使用方法和含义。作业 `transactionPolicyJob` 从整数中递增读取数据；`processor` 打印当前执行的数据，并根据属性 `errorCount` 的大小决定何时抛出 `RuntimeException` 异常；当异常 `RuntimeException` 抛出时，允许最大重试次数为 3 次；`writer` 负责将消息打印在控制台上。通过设置属性 `processor-transactional` 为不同的值，观察 `processor` 在控台上打印的消息：`processor-transactional="true"` 时，参见代码清单 5-29；`processor-transactional="false"` 时，参见代码清单 5-30。

代码清单 5-27 配置 `processor-transactional`

```
1.     <job id="transactionPolicyJob">
2.         <step id="transactionPolicyStep">
3.             <tasklet>
4.                 <chunk reader="reader" processor="processor" writer="writer"
5.                     commit-interval="5" processor-transactional="false"
6.                     retry-limit="3">
7.                     <retryable-exception-classes>
8.                         <include class="java.lang.RuntimeException" />
9.                     </retryable-exception-classes>
10.                </chunk>
11.            </tasklet>
12.        </step>
13.    </job>
14.    <bean:bean id="processor" class="com.juxtapose.example.ch05.
15.        TransactionItemProcessor">
16.        <bean:property name="errorCount" value="3"></bean:property>
17.    </bean:bean>
```

其中，5 行：设置提交间隔为 5，重试次数为 3 次，`processor` 不在事务中。

6~8 行：设置重试异常为 `java.lang.RuntimeException`。

13~15 行：定义任务处理器，并设置 `errorCount` 为 3 时抛出 `RuntimeException` 异常 `TransactionItemProcessor` 类声明，参见代码清单 5-28。

代码清单 5-28 类 `TransactionItemProcessor` 定义

```
1. public class TransactionItemProcessor implements ItemProcessor<String,
2.     String> {
3.     private String errorCount = "3";
4.     public String process(String item) throws Exception {
5.         System.out.println("TransactionItemProcessor.process() item is:"
6.             + item);
7.         if(errorCount.equals(item)){
8.             throw new RuntimeException("make error!");
9.         }
10.    }
```



```

8.         return item;
9.     }
10.    .....
11. }

```

其中，4 行：控制台输出当前正在处理的 item。

5~7 行：根据设定的 `errorCount` 抛出异常，`errorCount` 默认值为 3。

代码清单 5-29 `processor-transactional` 为 `true` 执行结果

```

1. TransactionItemProcessor.process() item is:1
2. TransactionItemProcessor.process() item is:2
3. TransactionItemProcessor.process() item is:3
4. TransactionItemProcessor.process() item is:1
5. TransactionItemProcessor.process() item is:2
6. TransactionItemProcessor.process() item is:3
7. TransactionItemProcessor.process() item is:1
8. TransactionItemProcessor.process() item is:2
9. TransactionItemProcessor.process() item is:3
10. TransactionItemProcessor.process() item is:1
11. TransactionItemProcessor.process() item is:2

```

当 `processor-transactional` 为 `true` 时，表示每次 `Chunk` 执行过程中，在处理的数据没有达到提交间隔之前，经过 `processor` 处理的 item 会放在缓存中；当发生了重试时候，会从缓存中第一条开始执行。本例中执行 Item 分别为 1、2、3，当 Item 为 3 的时候发生了异常导致进行重试操作，当 `processor-transactional` 为 `true` 的情况下进行重试的时候，会从 Item 值为 1 的开始重新执行，直到最后达到重试次数抛出异常终止 Job 为止。

代码清单 5-30 `processor-transactional` 为 `false` 执行结果

```

1. TransactionItemProcessor.process() item is:1
2. TransactionItemProcessor.process() item is:2
3. TransactionItemProcessor.process() item is:3
4. TransactionItemProcessor.process() item is:3
5. TransactionItemProcessor.process() item is:3

```

当 `processor-transactional` 为 `false` 时，表示每次 `Chunk` 执行过程中，经过 `processor` 处理的数据不会存放在缓存中，当发生了异常导致重试的时候，已经处理过的 1、2 两条记录不会再次被执行。请注意控制台输出，当第一次执行到记录 3 发生异常后，后面的重试操作均直接从记录 3 开始。

5.4 拦截器

`Chunk` 操作中提供了丰富的拦截器机制，通过拦截器可以实现额外的控制能力，例如日志记录、任务跟踪、状态报告、数据传递等能力；在使用 `Spring Batch` 的过程中，尽可能地保

持业务的简单性，任何额外的处理需要在拦截器中进行功能实现。

Chunk 操作中提供的拦截器，参见表 5-12。

表 5-12 Chunk 拦截器接口

Chunk 拦截器接口	功能说明
ChunkListener	批操作拦截器，主要操作：chunk 执行前，chunk 执行后 org.springframework.batch.core.ChunkListener
ItemProcessListener	处理器拦截器，主要操作：处理执行前，处理执行后，处理发生异常 org.springframework.batch.core.ItemProcessListener<T, S>
ItemReadListener	读操作拦截器，主要操作：读之前、读之后、读发生异常 org.springframework.batch.core.ItemReadListener<T>
ItemWriteListener	写操作拦截器，主要操作：写之前、写之后、写发生异常 org.springframework.batch.core.ItemWriteListener<S>
SkipListener	记录跳过拦截器，主要操作：读错误导致跳过时、写错误导致跳过时、处理错误导致跳过时 org.springframework.batch.core.SkipListener<T, S>
RetryListener	重试拦截器，主要操作：重试开始、重试结束、重试异常 org.springframework.batch.retry.RetryListener

Chunk 拦截器作用域参见图 5-14，按照作用域的大小分别执行对应的 before 和 after 操作，其中读、处理、写三个拦截器没有嵌套关系，三者按照顺序先后执行。Chunk 拦截器先后执行顺序参见代码清单 5-31。

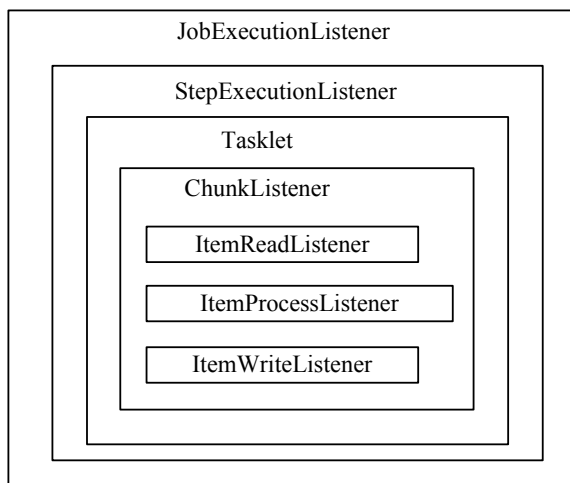


图 5-14 Chunk 拦截器作用域

代码清单 5-31 Chunk 拦截器先后执行顺序

```

1. JobExecutionListener.beforeJob()
2. StepExecutionListener.beforeStep()
3. ChunkListener.beforeChunk()
4. ItemReadListener.beforeRead()
5. ItemReadListener.afterRead()
6. ItemProcessListener.beforeProcess()
7. ItemProcessListener.afterProcess()
8. ItemWriteListener.beforeWrite()
9. ItemWriteListener.afterWrite()
10. ChunkListener.afterChunk()
11. StepExecutionListener.afterStep()
12. JobExecutionListener.afterJob()

```

典型的拦截器配置格式，参见代码清单 5-32。

代码清单 5-32 配置 Chunk 拦截器

```

1.     <job id="listenerJob">
2.         <step id="listenerStep">
3.             <tasklet>
4.                 <chunk reader="reader20" processor="processor" writer="writer"
5.                     commit-interval="1" retry-limit="1">
6.                     <retry-listeners>
7.                         <listener ref="retryListener"></listener>
8.                     </retry-listeners>
9.                     <retryable-exception-classes>
10.                        <include class="java.lang.RuntimeException" />
11.                    </retryable-exception-classes>
12.                    <listeners>
13.                        <listener ref="chunkListener"></listener>
14.                        <listener ref="itemReadListener"></listener>
15.                        <listener ref="itemProcessListener"></listener>
16.                        <listener ref="itemWriteListener"></listener>
17.                        <listener ref="skipListener"></listener>
18.                    </listeners>
19.                </chunk>
20.            </tasklet>
21.            <listeners>
22.                <listener ref="stepExecutionListener"></listener>
23.            </listeners>
24.        </step>
25.    </listeners>
26.        <listener ref="jobExecutionListener"></listener>
27.    </listeners>
28. </job>

```

其中，7 行：配置重试拦截器 `retryListener`，当发生指定的异常时会执行该拦截器操作。

13 行：配置批拦截器 `chunkListener`，在批处理操作执行时执行该拦截器。

14 行：配置读拦截器 `itemReadListener`，在批处理中的读操作执行时执行该拦截器。

15 行：配置处理拦截器 `itemProcessListener`，在批处理中的处理操作执行时执行该拦截器。

16 行：配置写拦截器 `itemWriteListener`，在批处理中的写操作执行时执行该拦截器。

17 行：配置跳过拦截器 `skipListener`，在批处理中当有记录被跳过时执行该拦截器。

22 行：配置 Step 执行的拦截器 `stepExecutionListener`，在批处理的每步执行时执行该拦截器。

26 行：配置 Job 执行的拦截器 `jobExecutionListener`，在批处理 Job 执行时执行该拦截器。

接下来本节会详细描述 `ChunkListener`、`SkipListener`、`RetryListener` 的具体使用，另外的读、处理、写阶段的拦截器在后面读、写、处理章节详细描述。

5.4.1 ChunkListener

批操作拦截器，主要操作：Chunk 执行前，Chunk 执行后。接口 `ChunkListener` 声明参见代码清单 5-33。

代码清单 5-33 `ChunkListener` 接口定义

```
1. public interface ChunkListener extends StepListener {
2.     void beforeChunk();
3.     void afterChunk();
4. }
```

`ChunkListener` 操作说明与 Annotation 定义，参见表 5-13。

表 5-13 `ChunkListener` 操作说明与 Annotation 定义

操 作	操作说明	Annotation
<code>beforeChunk()</code>	在 Chunk 执行前出发，在 Step 事务中	@ BeforeChunk
<code>afterChunk()</code>	在 Chunk 执行后出发，不在 Step 事务中	@ AfterChunk

`ChunkListener` 系统默认实现，参见表 5-14。

表 5-14 `ChunkListener` 默认实现

ChunkListener 默认实现	功能说明
<code>ChunkListenerSupport</code>	Chunk 拦截器的默认执行，可以继承该类仅实现关心的方法 <code>org.springframework.batch.core.listener.ChunkListenerSupport</code>
<code>CompositeChunkListener</code>	组合 Chunk 拦截器实现，可以定义一组的 Chunk 拦截器，依照顺序执行 <code>org.springframework.batch.core.listener.CompositeChunkListener</code>

5.4.2 ItemReadListener

`ItemReadListener` 在 `Chunk` 读阶段的拦截器，可以在读之前、读之后、读发生异常时候触发该拦截器。接口 `ItemReadListener` 声明参见代码清单 5-34。

代码清单 5-34 `ItemReadListener` 接口声明

```
1. public interface ItemReadListener<T> extends StepListener {
2.     void beforeRead();
3.     void afterRead(T item);
4.     void onError(Exception ex);
5. }
```

`ItemReadListener` 操作说明与 `Annotation` 定义，参见表 5-15。

表 5-15 `ItemReadListener` 操作说明与 `Annotation` 定义

操 作	操作说明	Annotation
<code>beforeRead()</code>	在 <code>ItemReader#read()</code> 之前执行	<code>@ BeforeRead</code>
<code>afterRead(T item)</code>	在 <code>ItemReader#read()</code> 之后执行	<code>@ AfterRead</code>
<code>onOnError(Exception ex)</code>	当 <code>ItemReader#read()</code> 抛出异常时候触发该操作	<code>@ OnReadError</code>

`ItemReadListener` 的具体使用，请参见 6.9 章节。

5.4.3 ItemProcessListener

`ItemProcessListener` 在 `Chunk` 处理阶段的拦截器，可以在处理之前、处理之后、处理发生异常时触发该拦截器。接口 `ItemProcessListener` 声明参见代码清单 5-35。

代码清单 5-35 `ItemProcessListener` 接口声明

```
1. public interface ItemProcessListener<T, S> extends StepListener {
2.     void beforeProcess(T item);
3.     void afterProcess(T item, S result);
4.     void onProcessError(T item, Exception e);
5. }
```

`ItemProcessListener` 操作说明与 `Annotation` 定义，参见表 5-16。

表 5-16 `ItemProcessListener` 操作说明与 `Annotation` 定义

操 作	操作说明	Annotation
<code>beforeProcess(T item)</code>	在 <code>ItemProcessor.process(Object)</code> 之前执行	<code>@ BeforeProcess</code>
<code>afterProcess(T item, S result)</code>	在 <code>ItemProcessor.process(Object)</code> 之后执行，即使 <code>process</code> 方法返回 <code>null</code> ，仍然会触发拦截器操作	<code>@ AfterProcess</code>
<code>onProcessError(T item, Exception e)</code>	当 <code>ItemProcessor.process(Object)</code> 抛出异常时触发该操作	<code>@ OnProcessError</code>

ItemProcessListener 的具体使用，请参见 8.7 章节。

5.4.4 ItemWriteListener

ItemWriteListener 在 Chunk 写阶段的拦截器，可以在写之前、写之后、写发生异常时触发该拦截器。接口 ItemWriteListener 声明参见代码清单 5-36。

代码清单 5-36 ItemWriteListener 接口声明

```
1. public interface ItemWriteListener<S> extends StepListener {
2.     void beforeWrite(List<? extends S> items);
3.     void afterWrite(List<? extends S> items);
4.     void onWriteError(Exception exception, List<? extends S> items);
5. }
```

ItemWriteListener 操作说明与 Annotation 定义参见表 5-17。

表 5-17 ItemWriteListener 操作说明与 Annotation 定义

操 作	操作说明	Annotation
beforeWrite(List<? extends S> items)	在 ItemWriter#write(java.util.List)之前执行	@ BeforeWrite
afterWrite(List<? extends S> items)	在 ItemWriter#write(java.util.List)之后执行 该操作会在事务提交之前、ChunkListener #afterChunk()之前执行	@ AfterWrite
onWriteError(Exception exception, List<? extends S> items)	当 ItemWriter#write(java.util.List)抛出异常时触发 该操作	@ OnWriteError

ItemWriteListener 的具体使用，请参见 7.12 章节。

5.4.5 SkipListener

SkipListener 在 Chunk 处理阶段抛出跳过定义的异常时触发，在 Chunk 的读、处理、写阶段发生的异常都会触发该拦截器。接口 SkipListener 声明参见代码清单 5-37。

代码清单 5-37 SkipListener 接口声明

```
1. public interface SkipListener<T,S> extends StepListener {
2.     void onSkipInRead(Throwable t);
3.     void onSkipInWrite(S item, Throwable t);
4.     void onSkipInProcess(T item, Throwable t);
5. }
```

SkipListener 操作说明与 Annotation 定义，参见表 5-18。

表 5-18 SkipListener 操作说明与 Annotation 定义

操 作	操作说明	Annotation
onSkipInRead(Throwable t)	在读阶段发生异常并且配置了异常可以跳过时触发该操作	@ OnSkipInRead
onSkipInWrite(S item, Throwable t)	在写阶段发生异常并且配置了异常可以跳过时触发该操作	@ OnSkipInWrite
onSkipInProcess(T item, Throwable t)	在处理阶段发生异常并且配置了异常可以跳过时触发该操作	@ OnSkipInProcess

SkipListener 系统默认实现类参见表 5-19。

表 5-19 SkipListener 默认实现

SkipListener 默认实现	功能说明
CompositeSkipListener	组合跳过拦截器实现，可以定义一组的跳过拦截器，依照顺序执行 org.springframework.batch.core.listener.CompositeSkipListener<T, S>
MulticasterBatchListener	同时实现 StepExecutionListener, ChunkListener, ItemReadListener<T>, ItemProcessListener<T, S>, ItemWriteListener<S>, SkipListener<T, S>六个接口组合策略的拦截器 org.springframework.batch.core.listener.MulticasterBatchListener<T, S>
SkipListenerSupport	跳过拦截器的默认执行，可以继承该类仅实现关心的方法 org.springframework.batch.core.listener.SkipListenerSupport<T, S>

SkipListener 的具体使用请参见 10 章节。

5.4.6 RetryListener

RetryListener 在 Chunk 处理阶段抛出重试定义的异常时触发，通过拦截器可以在重试动作发生时进行日志记录、收集重试信息等。可以直接实现接口 RetryListener，也可以直接继承 RetryListenerSupport，通常只需要实现 onError 操作，在重试发生错误时触发该操作。接口 RetryListener 声明参见代码清单 5-38。

代码清单 5-38 RetryListener 接口声明

```

1. public interface RetryListener {
2.     <T> boolean open(RetryContext context, RetryCallback<T> callback);
3.     <T> void close(RetryContext context, RetryCallback<T> callback,
4.         Throwable throwable);
5.     <T> void onError(RetryContext context, RetryCallback<T> callback,
6.         Throwable throwable);
7. }

```

RetryListener 操作说明参见表 5-20。

表 5-20 RetryListener 操作说明

操 作	操作说明	Annotation
open(RetryContext context, RetryCallback<T> callback)	在进入 retry 之前执行该操作，可以在该操作中准备重试需要的资源；如果该操作返回 false，将会终止本次重试操作，且会抛 出 异 常：org.springframework.batch.retry.Terminated RetryException	无
close(RetryContext context, RetryCallback<T> callback, Throwable throwable)	在 retry 结束之前执行该操作，可以在该方法中关闭在 open 操作中打开的资源	无
onError(RetryContext context, RetryCallback<T> callback, Throwable throwable)	重试发生错误时触发该操作	无

RetryListener 系统默认实现参见表 5-21。

表 5-21 RetryListener 默认实现

RetryListener 默认实现	功能说明
RetryListenerSupport	重试拦截器的默认执行，可以继承该类仅实现关心的方法 org.springframework.batch.retry.listener.RetryListenerSupport

RetryListener 的具体使用请参见 10 章节。

读数据 ItemReader

批处理通过 Tasklet 完成具体的任务，Chunk 类型的 Tasklet 定义了标准的读、处理、写的执行步骤。ItemReader 是实现读的重要组件，Spring Batch 框架提供了丰富的读基础设施来完成各种数据来源的读取功能，数据来源包括文本文件、Json 格式文件、XML 文件、DB、JMS 消息等。

Spring Batch 框架默认提供了丰富的 Reader 实现；如果不能满足需求可以快速方便地实现自定义的数据读取；对于已经存在的读服务，框架提供了复用现有服务的能力，避免重复开发。

Spring Batch 框架为保证 Job 的可靠性、稳定性，在读数据阶段提供了另外一个重要的特性是对读数据状态的保存，在 Spring Batch 框架内置的 Read 实现中通过与执行上下文进行读数据的状态交互，可以高效地保证 Job 的重启和错误处理。

6.1 ItemReader

ItemReader 是 Step 中对资源的读处理，Spring Batch 框架已经提供了各种类型的读实现，包括对文本文件、XML 文件、数据库、JMS 消息等读的处理。读操作与 Chunk Tasklet 的关系参见图 6-1。

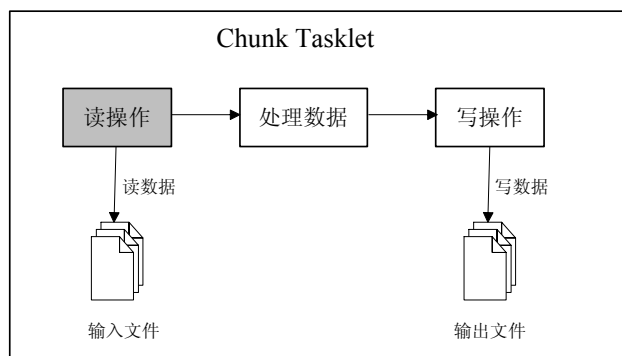


图 6-1 读操作与 Chunk Tasklet 关系图

6.1.1 ItemReader

所有的读操作需要实现 `org.springframework.batch.item.ItemReader<T>` 接口。ItemReader 接口定义参见代码清单 6-1。

代码清单 6-1 ItemReader 接口定义

```
1. public interface ItemReader<T> {
2.     T read() throws Exception,
3.         UnexpectedInputException, ParseException,
4.         NonTransientResourceException;
5. }
```

其中，2 行：ItemReader 接口定义了核心作业方法 read()操作，负责从给定的资源中读取可用的数据。

Job 中典型的配置 ItemReader 参见代码清单 6-2。

代码清单 6-2 配置 ItemReader

```
1. <job id="dbReadJob">
2.     <step id="dbReadStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="jdbcParameterItemReader"
5.                 processor="creditBillProcessor"
6.                 writer="creditItemWriter" commit-interval="2"></chunk>
7.         </tasklet>
8.     </step>
9. </job>
```

6.1.2 ItemStream

Spring Batch 框架同时提供了另外一个重要的接口 org.springframework.batch.item.ItemStream。ItemStream 接口定义参见代码清单 6-3。ItemStream 接口定义了读操作与执行上下文 ExecutionContext 交互的能力。可以将已经读的条数通过该接口存放在执行上下文 ExecutionContext 中（ExecutionContext 中的数据在批处理 commit 的时候会通过 JobRepository 持久化到数据库中），这样到 Job 发生异常重新启动 Job 的时候，读操作可以跳过已经成功读过的数据，继续从上次出错的地点（可以从执行上下文中获取上次成功读的位置）开始读。

代码清单 6-3 ItemStream 接口定义

```
1. public interface ItemStream {
2.     void open(ExecutionContext executionContext) throws
3.         ItemStreamException;
4.     void update(ExecutionContext executionContext) throws
5.         ItemStreamException;
6.     void close() throws ItemStreamException;
7. }
```

其中，2 行：open()操作根据参数 executionContext 打开需要读取资源的 stream；可以根据持久化在执行上下文 executionContext 中的数据重新定位需要读取记录的位置。

3 行：update()操作将需要持久化的数据存放在执行上下文 executionContext 中。

4 行：close()操作关闭读取的资源。

说明：Spring Batch 框架提供的读组件均实现了 `ItemStream` 接口。

6.1.3 系统读组件

Spring Batch 框架提供了丰富的读组件，包括对文件、关系数据库、NoSQL 数据库、消息队列等，具体参见表 6-1。

表 6-1 Spring Batch 框架提供的读组件

ItemReader	说 明
ListItemReader	读取 List 类型数据，只能读一次
ItemReaderAdapter	ItemReader 适配器，可以复用现有的读操作
FlatFileItemReader	读 Flat 类型文件
StaxEventItemReader	读 XML 类型文件
JdbcCursorItemReader	基于 JDBC 游标方式读数据库
HibernateCursorItemReader	基于 Hibernate 游标方式读数据库
StoredProcedureItemReader	基于存储过程读数据库
IbatisPagingItemReader	基于 Ibatis 分页读数据库
JpaPagingItemReader	基于 Jpa 方式分页读数据库
JdbcPagingItemReader	基于 JDBC 方式分页读数据库
HibernatePagingItemReader	基于 Hibernate 方式分页读取数据库
JmsItemReader	读取 JMS 队列
IteratorItemReader	迭代方式的读组件
MultiResourceItemReader	多文件读组件
MongoItemReader	基于分布式文件存储的数据库 MongoDB 读组件
Neo4jItemReader	面向网络的数据库 Neo4j 的读组件
ResourcesItemReader	基于批量资源的读组件，每次读取返回资源对象
AmqpItemReader	读取 AMQP 队列组件
RepositoryItemReader	基于 Spring Data 的读组件

6.2 Flat 格式文件

Flat 类型文件是一种包含没有相对关系结构的记录的文件。在批处理应用中经常需要处理的文件是简单文本格式文件，这类文件通常没有复杂的关系结构，通常经过分隔符分割，或者定长字段来描述数据格式；稍复杂的文件通过 JSON 的方式定义数据格式。

Spring Batch 框架提供的 `ItemReader` 本质上是从 Flat 文件中读取记录，并将记录转换为 Java 中的对象。在进行读取 Flat 文件之前，我们先学习一下 Flat 文件格式的几种不同类型。

6.2.1 Flat 文件格式

本章仍然采用信用卡对账单的例子来介绍 Flat 文件的格式。通过框架提供的对文件的 `ItemReader`，可以支持定长类型、分隔符类型、JSON 格式、跨多行分隔符、交替记录分隔符文件等。图 6-2 给出了 `ItemReader` 支持的多种文件类型。

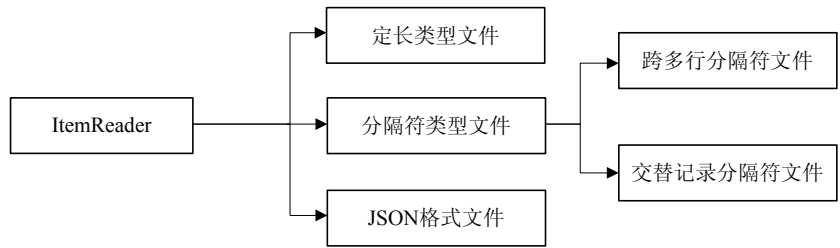


图 6-2 `ItemReader` 支持多种文件类型

6.2.1.1 分隔符类型文件

分隔符类型文件通常是根据设定的分隔符来区分一条记录中不同字段的，在每个字段中不能出现内容为分隔符的字符，否则会视为分隔符号。代码清单 6-4 给出了逗号分隔符的示例。

代码清单 6-4 逗号分隔符示例

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road
```

第一段表示信用卡号，第二段表示用户名，第三段表示消费金额，第四段表示消费日期，第五段表示消费地点。

6.2.1.2 定长类型文件

每一个字段的长度是确定的，定长类型的文件没有任何分隔符。代码清单 6-5 给出了定长类型文件的示例。

代码清单 6-5 定长类型文件示例

1.	4041390012345678	tom	00100.00	2013-02-02 12:00:08	Lu Jia Zui road
2.	4042390012345678	tom	00320.00	2013-02-03 10:35:21	Lu Jia Zui road
3.	4043390012345678	jerry	00674.70	2013-02-06 16:26:49	South Linyi road
4.	4044390012345678	rose	00793.20	2013-02-09 15:15:37	Longyang road
5.	4045390012345678	bruce	00360.00	2013-02-11 11:12:38	Longyang road
6.	4046390012345678	rachle	00893.00	2013-02-28 20:34:19	Hunan road

前 16 个字符表示信用卡卡号，接下来 12 个字符表示用户名，接下来 8 个字符表示消费金额，接下来 19 个字符表示消费日期，最后 16 个字符表示消费地点。

6.2.1.3 JSON 格式文件

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。它是基于 JavaScript (Standard ECMA-262 3rd Edition - December 1999) 的一个子集。

JSON 有两种结构: json 简单说就是 JavaScript 中的对象和数组, 所以这两种结构就是对象和数组, 通过这两种结构可以表示各种复杂的结构。

(1) 对象: 对象在 js 中表示为“{}”扩起来的内容, 数据结构为 {key: value,key: value,...} 的键值对的结构, 在面向对象的语言中, key 为对象的属性, value 为对应的属性值, 所以很容易理解, 取值方法为对象.key 获取属性值, 这个属性值的类型可以是数字、字符串、数组和对象。

(2) 数组: 数组在 js 中是用中括号“[]”扩起来的内容, 数据结构为["java","javascript","vb",...], 取值方式和所有语言中一样, 使用索引获取, 字段值的类型可以是数字、字符串、数组和对象。

经过对象、数组两种结构就可以组合成复杂的数据结构了。

代码清单 6-6 展示了信用卡数据是通过 JSON 格式进行存储的。

代码清单 6-6 JSON 格式数据示例

```
1. { "accountID": "4047390012345678",  
2.   "name": "tom",  
3.   "amount": 100.00,  
4.   "date": "2013-2-2 12:00:08",  
5.   "address": "Lu Jia Zui road"}  
6. { "accountID": "4047390012345678",  
7.   "name": "tom",  
8.   "amount": 320.00,  
9.   "date": "2013-2-3 10:35",  
10.  "address": "Lu Jia Zui road"}
```

6.2.1.4 记录跨多行格式

通常情况下一条记录占用一行, 在某些特殊情况下可能出现一条记录占用多行的场景, 下面展示了一条记录占用两行。第一行描述了信用卡编号、姓名、消费金额、消费时间, 第二行是消费地址。代码清单 6-7 给出了记录跨多行格式示例。

代码清单 6-7 记录跨多行格式示例

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08  
2. ,Lu Jia Zui road  
3. 4047390012345678,tom,320.00,2013-2-3 10:35:21  
4. ,Lu Jia Zui road  
5. 4047390012345678,tom,674.70,2013-2-6 16:26:49  
6. ,South Linyi road  
7. 4047390012345678,tom,793.20,2013-2-9 15:15:37  
8. ,Longyang road  
9. 4047390012345678,tom,360.00,2013-2-11 11:12:38
```

```

10. ,Longyang road
11. 4047390012345678,tom,893.00,2013-2-28 20:34:19
12. ,Hunan road

```

6.2.1.5 交替记录

另外一种更复杂的文件格式是在同一个文件中有两种或者更多类型的记录类型，不同的记录类型通过不同的前缀方式表示，下面展示了一个文件中两种不同类型的记录。代码清单 6-8 给出了交替记录的文件示例。

代码清单 6-8 交替记录的文件示例

```

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 3047390012345671,249.10,rose,2013/4/1 11:34:56
3. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
4. 3047390012345671,249.10,rose,2013/4/1 11:34:56
5. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
6. 3047390012345673,840.20,marry,2013/4/3 3:21:43
7. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road

```

以 40 开头的为信用卡消费记录，以 30 开头的为借记卡消费记录情况。

6.2.2 FlatFileItemReader

FlatFileItemReader 实现 ItemReader 接口，核心作用将 Flat 文件中的记录转换为 Java 对象。FlatFileItemReader 通过引用 RecordSeparatorPolicy、LineMapper、LineCallbackHandler 关键接口实现上面的目的；在 FlatFileItemReader 将记录转换为 Java 对象时主要有两步工作，首先根据 RecordSeparatorPolicy 从文件中确定一条记录，其次使用 LineMapper 将一条记录转换为 Java 对象，具体步骤参见图 6-3。

FlatFileItemReader 结构及关键属性

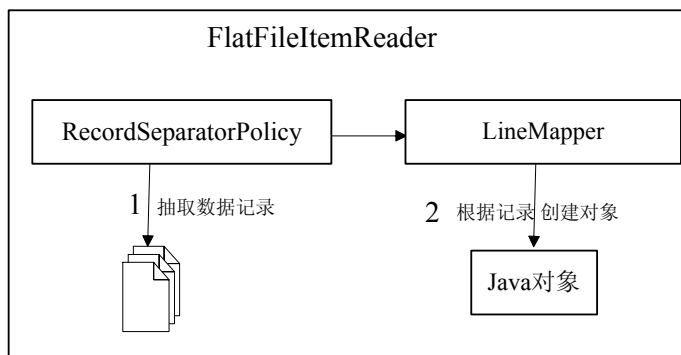


图 6-3 FlatFileItemReader 核心操作步骤

FlatFileItemReader 与关键接口 RecordSeparatorPolicy、LineMapper、LineCallbackHandler 之间的类图参见图 6-4，关键接口、类说明参见表 6-2。

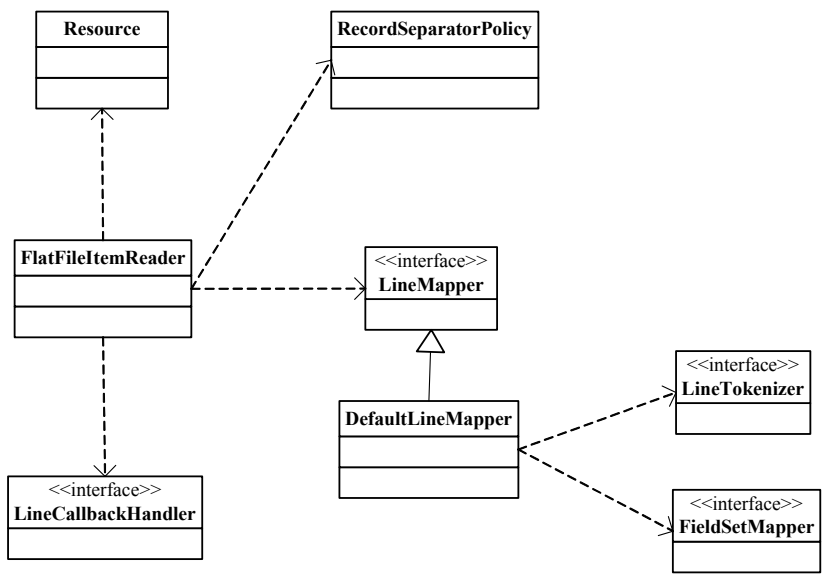


图 6-4 FlatFileItemReader 类关系图

FlatFileItemReader 引用 org.springframework.core.io.Resource，Resource 负责提供读取的资源信息，可以是文件，也可以是其他类型的资源；FlatFileItemReader 通过 RecordSeparatorPolicy 对文件进行分割，RecordSeparatorPolicy 定义了如何区分文件中的记录能力；LineMapper 对象提供最为核心的能力，将文件中的记录通过 LineMapper 对象转换为 Java 对象；LineCallbackHandler 在文件中行跳过时触发，通常与属性 linesToSkip 配合使用。

表 6-2 FlatFileItemReader 中关键接口、类说明

关 键 类	说 明
Resource	定义读取的文件资源
RecordSeparatorPolicy	从文件中确定一条记录的策略，记录可能是一行，可能是跨多行
LineMapper	将一条记录转化为 Java 数据对象，通常由 LineTokenizer 和 FieldSetMapper 组合来实现该功能
LineTokenizer	将一条记录分割为多个字段，在 LineMapper 的默认实现 DefaultLineMapper 中使用
FieldSetMapper	将多个字段值转化为 Java 对象，在 LineMapper 的默认实现 DefaultLineMapper 中使用
LineCallbackHandler	处理文件中记录回调处理操作

FlatFileItemReader 关键属性参见表 6-3。

表 6-3 FlatFileItemReader 关键属性

FlatFileItemReader 属性	类 型	说 明
bufferedReaderFactory	BufferedReaderFactory	根据给定的 resource 创建 BufferedReader 实例, 默认使用 DefaultBufferedReaderFactory 创建文本类型的 BufferedReader 实例
comments	String[]	定义注释行的前缀, 当某行以这些字符串中的一个开头时候, 此行记录将会被 Spring Batch 框架忽略
encoding	String	读取文件的编码类型, 默认值为从环境变量 file.encoding 获取, 如果没有设置则默认为 UTF-8
lineMapper	LineMapper<T>	将一行文件记录转换为 Java 对象
linesToSkip	int	读取文件时候, 定义跳过文件的行数; 跳过的行记录将会被传递给 skippedLinesCallback, 执行跳过行的回调操作
recordSeparatorPolicy	RecordSeparatorPolicy	定义文件如何区分记录, 可以按照单行、也可以按照多行区分记录
resource	Resource	需要读取的资源文件
skippedLinesCallback	LineCallbackHandler	定义文件中记录跳过时执行的回调操作, 通常与 linesToSkip 一起使用
strict	boolean	定义读取文件不存在时候的策略, 如果为 true 则抛出异常, 如果为 false 表示不抛出异常, 默认值为 true

配置 FlatFileItemReader

在给出详细配置文件之前, 我们首先给出一个更复杂的 csv 文件的复杂示例, 具体参见代码清单 6-9, 文件中增加了注释行, 注释以"##"或者"\$ \$"开头; 同时在头部有文件的结构性说明。

完整 CSV 格式文件参见: ch06/data/flat/credit-card-bill-201303-complex.csv。

代码清单 6-9 复杂 CSV 格式文件

```

1.  accountID,name,amount,date,address
2.  ## this line is first comment
3.  4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
4.  ## maybe address is not right
5.  4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
6.  $$ this line is comment, begin with &&
7.  4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
8.  4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
9.  4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
10. ## maybe time is not right
11. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road

```

其中, 1 行: 给出了文件的结构性说明, 该文件每条记录共有 5 个字段, 分别是信用卡账户 ID、姓名、消费金额、消费日期、消费地点; 在进行记录处理的时候需要跳过该行。

2、4、6、10 行：是注释信息，在进行信用卡消费对账处理的时候不希望处理这些记录。

了解了我们需要处理的 CSV 文件后，我们可利用现有的 FlatFileItemReader 组件，配置读复杂 CSV 格式文件，具体配置参见代码清单 6-10。

完整配置参见文件：ch06/job/job-flatfile.xml。

代码清单 6-10 配置 FlatFileItemReader

```
1.      <bean:bean id="flatFileItemReader" scope="step"
2.          class="org.springframework.batch.item.file.FlatFileItemReader">
3.          <bean:property name="resource"
4.              value="classpath:ch06/data/flat/credit-card-bill-201303-
                complex.csv"/>
5.          <bean:property name="lineMapper" ref="lineMapper" />
6.          <bean:property name="encoding" value="UTF-8" />
7.          <bean:property name="linesToSkip" value="1"/>
8.          <bean:property name="strict" value="true"/>
9.          <bean:property name="skippedLinesCallback" ref="lineCallbackHandler"/>
10.         <bean:property name="comments">
11.             <bean:list>
12.                 <bean:value>##</bean:value>
13.                 <bean:value>$$</bean:value>
14.             </bean:list>
15.         </bean:property>
16.         <bean:property name="recordSeparatorPolicy" ref="simpleRecord
                SeparatorPolicy" />
17.     </bean:bean>
18.     <bean:bean id="lineCallbackHandler"
19.         class="com.juxtapose.example.ch06.flat.DefaultLineCallback
                Handler">
20.     </bean:bean>
```

其中，3~4 行：定义读取的文件。

5 行：定义 lineMapper 属性，负责将单条记录转化为 Java 对象。

6 行：定义文件编码属性，使用 UTF-8 编码读取文件。

7 行：定义忽略文件的第一行，即文件第一行不处理，此处忽略的记录会触发属性 skippedLinesCallback 中定义的回调操作。

8 行：定义严格的文件存在检查策略，当 resource 中定义的文件不存在时会导致 Job 失败。

9 行：忽略文件前几行属性 linesToSkip 定义的情况下，这些忽略的记录被传递给此处定义的回调操作。

10~15 行：定义文件中的注释行，所有以"##"或者"\$\$"开头的行将会被忽略掉。

16 行：根据定义的策略 simpleRecordSeparatorPolicy 分割单条记录。

18~20 行：跳过行的回调操作，本实现中仅打印跳过的记录内容。

说明：在对 Flat 类型文件处理的时候，一行和一条记录是有区别的：一行内容不一定是一条记录，因为一条记录可能跨多行。

6.2.3 RecordSeparatorPolicy

RecordSeparatorPolicy 负责区分文件中不同记录的能力，即 RecordSeparatorPolicy 定义了如何从 Falt 文件中将不同的行记录区分开来作为一个完整的记录。通过该接口，读者可以实现任何复杂的文件记录区分的处理。

接口 org.springframework.batch.item.file.separator.RecordSeparatorPolicy 定义，参见代码清单 6-11。

代码清单 6-11 RecordSeparatorPolicy 接口定义

```
1. public interface RecordSeparatorPolicy {
2.     boolean isEndOfRecord(String line);
3.     String postProcess(String record);
4.     String preProcess(String record);
5. }
```

其中，isEndOfRecord()操作定义记录是否结束。

preProcess()操作在一个新行加入到记录前触发。

postProcess()操作在一个完整记录返回前触发。

RecordSeparatorPolicy 系统实现

Spring Batch 框架中提供了 4 类默认的实现，参见表 6-4，每种实现完成特定的记录分割能力，如果默认的 4 类实现不能满足要求，读者可以自行实现 RecordSeparatorPolicy 接口进行分割记录。

表 6-4 RecordSeparatorPolicy 系统默认实现类

RecordSeparatorPolicy	说 明
SimpleRecordSeparatorPolicy	每一行作为一条记录 org.springframework.batch.item.file.separator.SimpleRecordSeparatorPolicy
DefaultRecordSeparatorPolicy	如果行没有不匹配的引号，或者续行标识符"\n"，则每行作为一条记录，否则不作为一条记录；注意：判断行最后是否续行的时候，行后面的空格会被忽略掉 org.springframework.batch.item.file.separator.DefaultRecordSeparatorPolicy
SuffixRecordSeparatorPolicy	判断行是否以特定的后缀结束，如果是则认为是一条记录，否则不作为一条记录 org.springframework.batch.item.file.separator.SuffixRecordSeparatorPolicy
JsonRecordSeparatorPolicy	JSON 格式文件的行分割判断策略， org.springframework.batch.item.file.separator.JsonRecordSeparatorPolicy

说明：FlatFileItemReader 默认使用 SimpleRecordSeparatorPolicy 作为记录分割策略。

6.2.4 LineMapper

LineMapper 接口定义了如何将 Flat 文件中的一条记录转化为领域对象（通常为 Java 对象）。LineMapper 接口仅有一个方法 mapLine(), 传入的参数是行的内容和行号，返回值为转换后的领域对象。LineMapper 接口定义参见代码清单 6-12。

代码清单 6-12 LineMapper 接口定义

```
1. public interface LineMapper<T> {  
2.     T mapLine(String line, int lineNumber) throws Exception;  
3. }
```

Spring Batch 框架中提供了 4 类默认的实现，参见表 6-5，每种实现完成特定的数据转换，如果默认的 4 类实现不能满足要求，读者可以自行定义 LineMapper 接口进行数据转换。

表 6-5 LineMapper 系统默认实现类

LineMapper	说 明
DefaultLineMapper<T>	默认的行转换类，引用接口 LineTokenizer 和 FieldSetMapper<T>完成数据转换；LineTokenizer 负责将一条记录转换为对象 FieldSet（可以看作是一个 key-value 对的组合），FieldSetMapper<T>负责将 FieldSet 转换为领域对象 org.springframework.batch.item.file.mapping.DefaultLineMapper<T>
JsonLineMapper	负责将文件中 JSON 格式的文本数据转换为领域对象，转换后的领域对象格式为 Map<String, Object> org.springframework.batch.item.file.mapping.JsonLineMapper
PassThroughLineMapper	最简化的数据转换实现类，将一条记录直接返回，可以认为返回的领域对象为 String 类型的格式 org.springframework.batch.item.file.mapping.PassThroughLineMapper
PatternMatchingCompositeLineMapper<T>	复杂数据转换类，可以为不同的记录定义不同的 LineTokenizer 和 FieldSetMapper<T>来实现数据转换；在执行过程中，根据每条记录的内容根据设置的条件找到匹配的 LineTokenizer 和 FieldSetMapper<T>进行数据转换；多用于处理同一文件中有不同类型记录的场景 可以认为 PatternMatchingCompositeLineMapper<T>组合了多个 DefaultLineMapper<T> org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper<T>

接下来的章节详细介绍 DefaultLineMapper 的功能实现。

6.2.5 DefaultLineMapper

org.springframework.batch.item.file.mapping.DefaultLineMapper<T>默认使用 LineTokenizer 和 FieldSetMapper<T>完成数据转换功能。DefaultLineMapper 通过组合的方式将任务委托给两个接口来完成，简化了 DefaultLineMapper 的代码结构，同时使得每个对象完成的任务职责非常简单，保持了代码结构清晰。DefaultLineMapper、LineTokenizer、FieldSetMapper 三者之间的关系参见图 6-5。

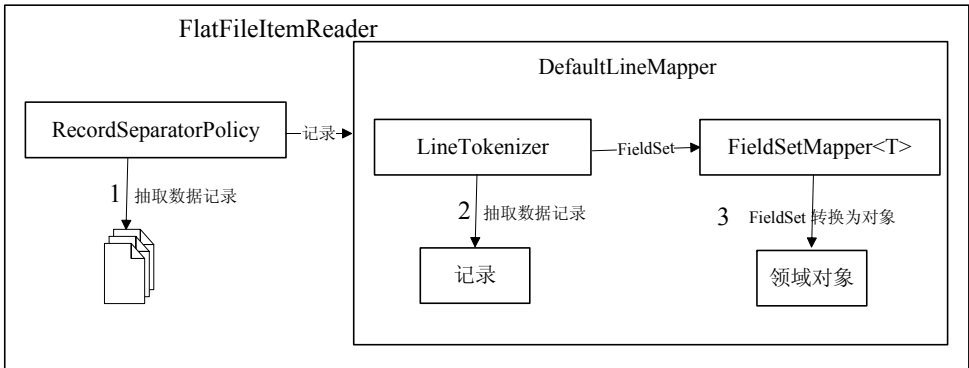


图 6-5 DefaultLineMapper、LineTokenizer、FieldSetMapper 三者之间的关系

其中，LineTokenizer 负责将一条记录转换为 FieldSet 对象；
FieldSetMapper 负责将 FieldSet 对象转换为领域对象。

代码清单 6-13 给出了配置 DefaultLineMapper 示例代码。

代码清单 6-13 配置 DefaultLineMapper

```
1. <bean:bean id="lineMapper"
2.     class="org.springframework.batch.item.file.mapping.
        DefaultLineMapper" >
3.     <bean:property name="lineTokenizer" ref=" delimitedLineTokenizer " />
4.     <bean:property name="fieldSetMapper"
        ref="creditBillBeanWrapperFieldSetMapper"/>
5. </bean:bean>
```

其中，3 行：属性 lineTokenizer，使用 DelimitedLineTokenizer 将行记录转换为 FieldSet 对象。

4 行：属性 fieldSetMapper，使用 BeanWrapperFieldSetMapper 将 FieldSet 对象转换为领域对象 CreditBill。

完成的配置 DefaultLineMapper

代码清单 6-14 给出了完成的配置 LineMapper，使用了 DelimitedLineTokenizer 和 BeanWrapperFieldSetMapper 进行数据转换。

代码清单 6-14 配置 LineMapper

```
1.     <bean:bean id="lineMapper"
2.         class="org.springframework.batch.item.file.mapping.
3.             DefaultLineMapper" >
4.         <bean:property name="lineTokenizer" ref="delimitedLineTokenizer" />
5.         <bean:property name="fieldSetMapper"
6.             ref="creditBillBeanWrapperFieldSetMapper"/>
7.     </bean:bean>
8.
9.     <bean:bean id="delimitedLineTokenizer"
10.        class="org.springframework.batch.item.file.transform.
11.            DelimitedLineTokenizer">
12.        <bean:property name="delimiter" value="," />
13.        <bean:property name="names" value="accountID,name,amount,date,
14.            address" >
15.    </bean:bean>
16.
17.    <bean:bean id="creditBillBeanWrapperFieldSetMapper"
18.        class="org.springframework.batch.item.file.mapping.
19.            BeanWrapperFieldSetMapper" >
20.        <bean:property name="prototypeBeanName" value="creditBill" />
21.    </bean:bean>
```

接下来详细介绍接口 `LineTokenizer` 与 `FieldSetMapper<T>`。

6.2.5.1 LineTokenizer

`LineTokenizer` 接口负责将一条记录转换 `FieldSet` 对象。接口仅有一个方法 `tokenize()`，传入的参数是行的内容，返回值为 `FieldSet` 对象。`FieldSet` 对象中可以认为是一组 `key-value` 的组合，负责存放每条记录分割后的数据条目，条目以 `key-value`（`key` 可以认为是字段的名称 `name`）的方式存在。`FieldSet` 接口中定义了读取 `value` 值的基本类型操作，以及操作 `name` 相关的方法。`LineTokenizer` 接口定义参见代码清单 6-15。`FieldSet` 接口定义参见代码清单 6-16。

代码清单 6-15 LineTokenizer 接口定义

```
1. public interface LineTokenizer {
2.     FieldSet tokenize(String line);
3. }
```

代码清单 6-16 FieldSet 接口定义

```
1. public interface FieldSet {
2.     String[] getNames();
3.     boolean hasNames();
4.     String[] getValues();
5.     String readString(int index);
6.     String readString(String name);
```

```

7.    String readRawString(int index);
8.    String readRawString(String name);
9.    boolean readBoolean ...;
10.   char readChar ...;
11.   byte readByte ...;
12.   short readShort ...;
13.   int readInt ...;
14.   long readLong ...;
15.   float readFloat ...;
16.   double readDouble ...;
17.   BigDecimal readBigDecimal ...;
18.   Date readDate ..;
19.   int getFieldCount();
20.   Properties getProperties();
21. }

```

在 FieldSet 中读取 value 时，支持直接转换为类型 String、Boolean、Char、Byte、Short、Int、Long、Float、Double、BigDecimal、Date。

LineTokenizer 系统实现

Spring Batch 框架中提供了 4 类默认的实现，参见表 6-6。每种实现完成特定的记录转换，如果默认的 4 类实现不能满足要求，读者可以自行定义 LineTokenizer 接口进行数据转换。

表 6-6 LineTokenizer 系统默认实现

LineTokenizer	说 明
DelimitedLineTokenizer	基于分隔符的行转换，根据给定的分隔符将一条记录转换为 FieldSet 对象 org.springframework.batch.item.file.transform.DelimitedLineTokenizer
FixedLengthTokenizer	基于定长数据的行转换，根据给定的数据长度将一条记录转换为 FieldSet 对象 org.springframework.batch.item.file.transform.FixedLengthTokenizer
RegexLineTokenizer	根据正则表达式的条件将一条记录转换为 FieldSet 对象 org.springframework.batch.item.file.transform.RegexLineTokenizer
PatternMatchingComposite LineTokenizer	可以为不同的记录定义不同的 LineTokenizer；在执行过程中根据给定的标识与每条记录比对，如果满足则用指定的 LineTokenizer；多用于处理同一文件中有不同类型记录的场景 org.springframework.batch.item.file.transform.PatternMatchingCompositeLineTokenizer

LineTokenizer 配置文件

接下来给读者介绍分隔符和定长类型的文件处理。DelimitedLineTokenizer 用来处理分隔符文件，FixedLengthTokenizer 用来处理定长文件。

分隔符文件处理

代码清单 6-17 给出了需要处理的分隔符文件，每行记录用逗号分隔，通过 `DelimitedLineTokenizer` 可以快速地将每个字段和名称进行映射，代码清单 6-18 给出了如何配置 `DelimitedLineTokenizer`。

代码清单 6-17 待处理分隔符文件

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road
```

分隔符 `DelimitedLineTokenizer` 的配置信息如代码清单 6-18 所示。

代码清单 6-18 配置分隔符 `DelimitedLineTokenizer`

```
1. <bean:bean id="delimitedLineTokenizer"
2.     class="org.springframework.batch.item.file.transform.
    DelimitedLineTokenizer">
3.     <bean:property name="delimiter" value=","/>
4.     <bean:property name="names" >
5.         <bean:list>
6.             <bean:value>accountID</bean:value>
7.             <bean:value>name</bean:value>
8.             <bean:value>amount</bean:value>
9.             <bean:value>date</bean:value>
10.            <bean:value>address</bean:value>
11.        </bean:list>
12.    </bean:property>
13. </bean:bean>
```

其中，3 行：属性 `delimiter` 定义分隔符标识为逗号","分割的字段内容按照顺序和 `names` 定义的名字匹配。

4~12 行：属性 `names` 为每个字段定义名字，根据分隔符","获取的每个字段值分别对应 `names` 中定义的名字。

以第一行数据为例子，描述分隔符字段与 `name` 之间的匹配关系，参见表 6-7。

表 6-7 分隔符字段与 `name` 的配置关系

name	行中的字段值
accountID	4047390012345678
name	tom
amount	100.00
date	2013-2-2 12:00:08
address	Lu Jia Zui road

定长类型文件处理

代码清单 6-19 给出了需要处理的定长文件，每行记录长度一致，字段间不足的使用空格补齐，每个字段的固定长度为 1-16,17-26,27-34,35-53,54-72。通过 `FixedLengthTokenizer` 可以快速地每个字段和名称进行映射，代码清单 6-20 给出了如何配置 `FixedLengthTokenizer`。

代码清单 6-19 待处理定长类型文件

```
1. 4041390012345678tom      00100.002013-02-02 12:00:08  Lu Jia Zui road
2. 4042390012345678tom      00320.002013-02-03 10:35:21  Lu Jia Zui road
3. 4043390012345678jerry    00674.702013-02-06 16:26:49  South Linyi road
4. 4044390012345678rose     00793.202013-02-09 15:15:37  Longyang road
5. 4045390012345678bruce    00360.002013-02-11 11:12:38  Longyang road
6. 4046390012345678rachle   00893.002013-02-28 20:34:19  Hunan road
```

代码清单 6-20 配置定长文件 `FixedLengthTokenizer`

```
1. <bean:bean id="fixedLengthLineTokenizer"
2.     class="org.springframework.batch.item.file.transform.
      FixedLengthTokenizer">
3.     <bean:property name="columns" value="1-16,17-26,27-34,35-53,
      54-72"/>
4.     <bean:property name="names" value="accountID,name,amount,date,
      address"/>
5. </bean:bean>
```

其中，3 行：属性 `columns` 定义字段长度，字段顺序和 `names` 定义的名字匹配。

4 行：属性 `names` 为每个字段定义名字。

以第一行数据为例子，描述定长字段与 `name` 之间的匹配关系，参见表 6-8。

表 6-8 定长字段与 `name` 的配置关系

name	行中的字段值	长度范围
accountID	4047390012345678	1-16
name	tom	17-26
amount	100.00	27-34
date	2013-2-2 12:00:08	35-53
address	Lu Jia Zui road	54-72

说明：定义字段长度时，从 1 开始，而不是从 0 开始。

6.2.5.2 FieldSetMapper

`FieldSetMapper` 接口定义了如何将 `FieldSet` 对象转化为领域对象（通常为 Java 对象）。`FieldSetMapper` 接口仅有一个方法 `mapFieldSet()`，传入的参数是 `FieldSet` 对象，返回值为转换后的领域对象。`FieldSetMapper` 接口定义参见代码清单 6-21。

代码清单 6-21 FieldSetMapper 接口定义

```
1. public interface FieldSetMapper<T> {
2.     T mapFieldSet(FieldSet fieldSet) throws BindException;
3. }
```

FieldSetMapper 系统实现

Spring Batch 框架中提供了 3 类默认的实现，参见表 6-9，每种实现完成特定的数据转换，如果默认的 3 类实现不能满足要求，读者可以自行实现 FieldSetMapper 接口进行数据转换。

表 6-9 FieldSetMapper 系统默认实现

FieldSetMapper	说 明
ArrayFieldSetMapper	将 FieldSet 对象转换为 String[] org.springframework.batch.item.file.mapping.ArrayFieldSetMapper
BeanWrapperFieldSetMapper<T>	将 FieldSet 对象根据名字映射到给定的 Bean 中，需要保证 FieldSet 中的 name 和 Bean 中属性的名称一致 org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper<T>
PassThroughFieldSetMapper	直接返回 FieldSet 对象 org.springframework.batch.item.file.mapping.PassThroughFieldSetMapper

接下来，我们向读者介绍如何使用 BeanWrapperFieldSetMapper 和自定义的 FieldSetMapper。

BeanWrapperFieldSetMapper

BeanWrapperFieldSetMapper 提供了一种方便的方式，将 FieldSet 对象根据名字直接映射成领域对象。如果声明使用 BeanWrapperFieldSetMapper，只需要在属性 prototypeBeanName 中指明需要映射的领域对象即可，在运行期自动将 FieldSet 中 name 与领域对象同名的属性进行赋值操作。

BeanWrapperFieldSetMapper 的配置信息，参见代码清单 6-22。

代码清单 6-22 配置 BeanWrapperFieldSetMapper

```
1. <bean:bean id="creditBillBeanWrapperFieldSetMapper"
2.     class="org.springframework.batch.item.file.mapping.
3.         BeanWrapperFieldSetMapper" >
4.     <bean:property name="prototypeBeanName" value="creditBill" />
5. </bean:bean>
6. <bean:bean id="creditBill" scope="prototype"
7.     class="com.juxtapose.example.ch06.CreditBill">
8. </bean:bean>
```

其中，1 行：定义 BeanWrapperFieldSetMapper，将 FieldSet 映射到 creditBill 对象中。

3 行：属性 prototypeBeanName，指定需要映射的领域对象。

5~7 行：定义信用卡对账单类 `CreditBill`，`CreditBill` 对象中的属性和 `FieldSet` 中的 `name` 保持一致，`BeanWrapperFieldSetMapper` 能够自动根据 `name` 属性将 `FieldSet` 中的 `value` 值映射到 `CreditBill` 对象中。

类 `com.juxtapose.example.ch06.CreditBill` 的定义参见代码清单 6-23。

代码清单 6-23 类 `CreditBill` 定义

```
1. public class CreditBill {
2.     private String accountID = "";    /** 银行卡账户 ID */
3.     private String name = "";        /** 持卡人姓名 */
4.     private double amount = 0;       /** 消费金额 */
5.     private String date;             /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */
6.     private String address;          /** 消费场所 */
7. }
```

在实际的任务开发过程中，可以实现自定义的 `FieldSetMapper`，将 `FieldSet` 对象转换成实际需要的领域对象。接下来我们向读者介绍如何使用自定义的 `FieldSetMapper`。

自定义 `FieldSetMapper`

直接实现接口 `FieldSetMapper`，可以实现自定义的转换器，下面的代码将 `FieldSet` 对象直接转换为领域对象 `CreditBill`。

自定义转换器 `CreditBillFieldSetMapper` 的代码参见代码清单 6-24。

代码清单 6-24 自定义转换器 `CreditBillFieldSetMapper` 类

```
1. public class CreditBillFieldSetMapper implements
   FieldSetMapper <CreditBill> {
2.     public CreditBill mapFieldSet(FieldSet fieldSet) throws BindException {
3.         CreditBill result = new CreditBill();
4.         result.setAccountID(fieldSet.readString("accountID"));
5.         result.setName(fieldSet.readString("name"));
6.         result.setAmount(fieldSet.readDouble("amount"));
7.         result.setDate(fieldSet.readString("date"));
8.         result.setAddress(fieldSet.readString("address"));
9.         return result;
10.    }
11. }
```

唯一的实现方法 `mapFieldSet()`，在操作中将 `fieldSet` 对象内的值转换为领域对象 `CreditBill`。代码中使用了 `FieldSet` 中的 `readXXX()` 操作，在 `readXXX()` 操作中传入 `name` 参数获取基本类型数据。

开发完毕自定义转化器 `CreditBillFieldSetMapper` 后，配置该类非常简单，只需要声明该 `Bean` 即可。配置文件参见代码清单 6-25。

代码清单 6-25 配置自定义转换器 CreditBillFieldSetMapper

```
1. <bean:bean id="creditBillFieldSetMapper"
2.     class="com.juxtapose.example.ch06.flat.CreditBillFieldSetMapper">
3. </bean:bean>
```

6.2.6 LineCallbackHandler

在 FlatFileItemReader 中定义属性 linesToSkip，表示从文件头开始跳过指定的行；当记录被跳过时，会触发接口 LineCallbackHandler 中的 handleLine()操作，该操作中能够获取跳过的行内容。

经常使用的场景是将 Flat 文件中的文件头跳过后，直接写到目标文件的文件头中，这可以通过实现 LineCallbackHandler 接口来完成。LineCallbackHandler 接口定义参见代码清单 6-26。

代码清单 6-26 LineCallbackHandler 接口定义

```
1. public interface LineCallbackHandler {
2.     void handleLine(String line);
3. }
```

说明：操作 handleLine()在每次跳过行时均会被触发。

自定义实现 LineCallbackHandler

实现接口 LineCallbackHandler，将跳过的文件内容写入到目标文件中。

com.juxtapose.example.ch06.flat.CopyHeaderLineCallbackHandler 的实现，参见代码清单 6-27。

代码清单 6-27 类 CopyHeaderLineCallbackHandler 定义

```
1. public class CopyHeaderLineCallbackHandler implements LineCallbackHandler,
2.     FlatFileHeaderCallback {
3.     private String header = "";
4.
5.     public void handleLine(String line) {
6.         this.header = line;
7.     }
8.
9.     public void writeHeader(Writer writer) throws IOException {
10.        writer.write(header);
11.    }
12. }
```

其中，CopyHeaderLineCallbackHandler 同时实现了接口 LineCallbackHandler 和 FlatFileHeaderCallback。接口 FlatFileHeaderCallback 用于定义在 ItemWrite 之前写入特定的文件头，主要的操作是 writeHeader()。

handleLine()操作负责收集读文件跳过的头记录，writeHeader()操作负责写入到目标文件中。

配置文件 `ch06/job/job-flatfile-copyheader.xml` 展示了如何配置回调操作，参见代码清单 6-28。

代码清单 6-28 `job-flatfile-copyheader.xml`

```
1.     <bean:bean id="copyHeaderItemReader" scope="step"  
2.         class="org.springframework.batch.item.file.FlatFileItemReader">  
3.         <bean:property name="linesToSkip" value="1"/>  
4.         <bean:property name="skippedLinesCallback"  
5.             ref="copyHeaderLineCallbackHandler"/>  
6.         .....  
7.     </bean:bean>  
8.     <bean:bean id="copyHeaderLineCallbackHandler"  
9.         class="com.juxtapose.example.ch06.flat.  
10.             CopyHeaderLineCallbackHandler">  
11.     </bean:bean>  
12.     <bean:bean id="csvItemWriter"  
13.         class="org.springframework.batch.item.file.FlatFileItemWriter" >  
14.         <bean:property name="headerCallback"  
15.             ref="copyHeaderLineCallbackHandler"/>  
16.         .....  
17.     </bean:bean>
```

其中，3 行：定义跳过的行数为 1 行。

4 行：定义跳过的记录的触发器为 `copyHeaderLineCallbackHandler`。

12 行：定义写文件之前，先写入特定的文件头信息。

查看执行后的文件（`target/ch06/copyheader/outputFile.csv`），参见代码清单 6-29，头文件已经成功写入。

代码清单 6-29 写入后的文件

```
1.  accountID,name,amount,date,address  
2.  4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road  
3.  .....
```

截至本节，读 Flat 文件所用到的核心类和接口基本介绍完毕，后面我们将给出读取不同类型文件的实际使用场景，包括读分隔符文件、读定长文件、读 JSON 文件、读记录跨多行文件、读混合记录文件，在每节将会有完整的示例展示如何使用上面的核心类和接口。

6.2.7 读分隔符文件

分隔符文件使用 Spring Batch 框架提供的默认组件即可轻松地配置完成，使用到的核心组件包括 `DelimitedLineTokenizer`、`CreditBillFieldSetMapper`（自定义实现）；其他没有在图 6-6 中体现的均使用了 `FlatFileItemReader` 的默认属性。

读分隔符文件使用的核心组件类参见图 6-6。

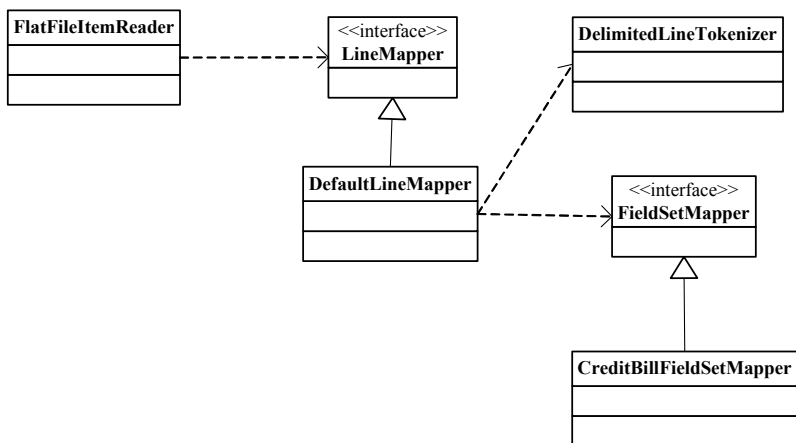


图 6-6 读分隔符文件使用的核心组件类

DelimitedLineTokenizer: 根据分隔符将一条记录转换为 **FieldSet** 对象。

CreditBillFieldSetMapper: 自定义的 **FieldSetMapper**，将 **FieldSet** 对象直接映射为 **CreditBill**。

自定义转换器 **CreditBillFieldSetMapper** 的代码，参见代码清单 6-30。

代码清单 6-30 自定义转换器 **CreditBillFieldSetMapper** 类定义

```

1. public class CreditBillFieldSetMapper implements
   FieldSetMapper <CreditBill> {
2.     public CreditBill mapFieldSet(FieldSet fieldSet) throws BindException {
3.         CreditBill result = new CreditBill();
4.         result.setAccountID(fieldSet.readString("accountID"));
5.         result.setName(fieldSet.readString("name"));
6.         result.setAmount(fieldSet.readDouble("amount"));
7.         result.setDate(fieldSet.readString("date"));
8.         result.setAddress(fieldSet.readString("address"));
9.         return result;
10.    }
11. }
  
```

分隔符格式文件读取配置，参见代码清单 6-31。

代码清单 6-31 配置读分隔符格式文件

```

1. <bean:bean id="flatFileItemReader" scope="step"
2.     class="org.springframework.batch.item.file.FlatFileItemReader">
3.     <bean:property name="resource"
4.         value="classpath:ch06/data/flat/credit-card-bill-201303.csv"/>
5.     <bean:property name="lineMapper" ref="lineMapper" />
6. </bean:bean>
7.
  
```

```

8.      <bean:bean id="lineMapper"
9.          class="org.springframework.batch.item.file.mapping.
              DefaultLineMapper" >
10.         <bean:property name="lineTokenizer" ref="delimitedLineTokenizer" />
11.         <bean:property name="fieldSetMapper" ref="creditBillFieldSetMapper"/>
12.     </bean:bean>
13.
14.     <bean:bean id="delimitedLineTokenizer"
15.         class="org.springframework.batch.item.file.transform.
              DelimitedLineTokenizer">
16.         <bean:property name="delimiter" value="," />
17.         <bean:property name="names" value="accountID,name,amount,date,
              address" />
18.     </bean:bean>
19.
20.     <bean:bean id="creditBillFieldSetMapper"
21.         class="com.juxtapose.example.ch06.flat.
              CreditBillFieldSetMapper">
22.     </bean:bean>

```

其中，1~6行：定义 `ItemReader` 类，主要包括需要读取的资源 and 行数据转换类。

8~12行：定义行数据转换类 `DefaultLineMapper`，使用分隔符类和自定义的 `FieldSetMapper`。

14~18行：定义 `DelimitedLineTokenizer`，属性 `delimiter` 字段区分标志，属性 `names` 定义字段映射的名字。

20~22行：声明自定义的 `CreditBillFieldSetMapper`。

6.2.8 读定长文件

定长格式文件使用 Spring Batch 框架提供的默认组件即可轻松地配置完成，使用的核心组件包括 `FixedLengthTokenizer`、`BeanWrapperFieldSetMapper`；其他没有在图 6-7 中体现的均使用了 `FlatFileItemReader` 的默认属性。

读定长文件使用的核心组件类图参见图 6-7。

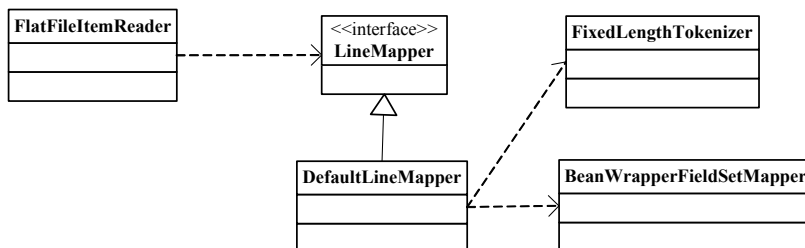


图 6-7 读定长文件使用的核心组件类

其中, FixedLengthTokenizer: 根据字段长度将一条记录转换为 FieldSet 对象。
BeanWrapperFieldSetMapper: 负责将 FieldSet 对象根据 name 映射到指定的 Java Bean 中。
定长格式文件读取配置, 参见代码清单 6-32。

代码清单 6-32 配置读定长格式文件

```
1. <bean:bean id="fixedLengthItemReader">
2.     class="org.springframework.batch.item.file.FlatFileItemReader">
3.     <bean:property name="resource"
4.         value="classpath:ch06/data/flat/credit-card-bill-fixed-length-
           201303.csv"/>
5.     <bean:property name="lineMapper" ref="fixedLengthLineMapper"/>
6. </bean:bean>
7.
8. <bean:bean id="fixedLengthLineMapper">
9.     class="org.springframework.batch.item.file.mapping.DefaultLine
       Mapper">
10.    <bean:property name="lineTokenizer" ref="fixedLengthLineTokenizer"/>
11.    <bean:property name="fieldSetMapper" ref="creditBillBeanWrapper
       FieldSetMapper"/>
12. </bean:bean>
13.
14. <bean:bean id="fixedLengthLineTokenizer">
15.     class="org.springframework.batch.item.file.transform.
       FixedLengthTokenizer">
16.     <bean:property name="columns" value="1-16,17-26,27-34,35-53,54-72"/>
17.     <bean:property name="names" value="accountID,name,amount,date,
       address"/>
18. </bean:bean>
19.
20. <bean:bean id="creditBillBeanWrapperFieldSetMapper">
21.     class="org.springframework.batch.item.file.mapping.
       BeanWrapperFieldSetMapper" >
22.     <bean:property name="prototypeBeanName" value="creditBill" />
23. </bean:bean>
```

其中, 1~6 行: 定义 ItemReader 类, 主要包括需要读取的资源 and 行数据转换类。

8~12 行: 定义行数据转换类 DefaultLineMapper, 使用定长的分隔符类和 Bean 自动装配的 FieldSetMapper。

14~18 行: 定义 FixedLengthTokenizer, 属性 columns 主要定义每个字段的长度, 属性 names 定义字段映射的名字。

20~23 行: 属性 prototypeBeanName 指定了映射的 Java Bean 对象, 通过 name 将 FieldSet 中的 value 自动映射到 creditBill 对象上同名的属性上。

6.2.9 读 JSON 文件

JSON 格式数据在 Java 领域使用非常广泛，使用 Spring Batch 框架提供的默认实现可以方便地实现对 JSON 格式文件的读取。读取 JSON 文件使用到的核心组件类参见图 6-8。

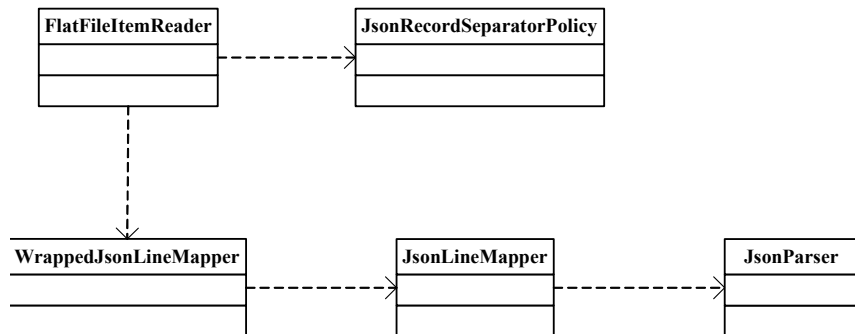


图 6-8 读取 JSON 文件使用到的核心组件类

JsonRecordSeparatorPolicy: Spring Batch 框架提供，根据匹配的“}”分割 JSON 格式文件，每个完整且匹配的部分被认为是一条记录。代码清单 6-33 中的 JSON 文件经过 **JsonRecord SeparatorPolicy** 拆分后，1~5 行被组装为一条记录，6~10 行被组装为另外一条记录。

代码清单 6-33 JSON 格式文件

```
1. { "accountID": "4047390012345678",
2.   "name": "tom",
3.   "amount": 100.00,
4.   "date": "2013-2-2 12:00:08",
5.   "address": "Lu Jia Zui road"}
6. { "accountID": "4047390012345678",
7.   "name": "tom",
8.   "amount": 320.00,
9.   "date": "2013-2-3 10:35",
10.  "address": "Lu Jia Zui road"}
```

其中，**JsonLineMapper**: Spring Batch 框架提供将一条完成的 JSON 记录转换为 **Map<String, Object>** 对象。在实际使用 **JsonLineMapper** 时候，可以继续封装 **JsonLineMapper**，在默认返回 **Map<String, Object>** 的基础上封装返回特定的领域对象。

WrappedJsonLineMapper: 扩展实现，用于代理 **JsonLineMapper**，负责将一条记录直接转换为领域对象 **CreditBill**。**WrappedJsonLineMapper** 将转换工作委托给 **JsonLineMapper**，并将 **Map** 结构转换为 **CreditBill**。

代码清单 6-34 展示了 **WrappedJsonLineMapper** 的实现。

代码清单 6-34 WrappedJsonLineMapper 类定义

```
1. public class WrappedJsonLineMapper implements LineMapper<CreditBill> {
2.     private JsonLineMapper delegate;
3.
4.     public CreditBill mapLine(String line, int lineNumber) throws Exception {
5.         CreditBill result = new CreditBill();
6.         Map<String, Object> creditBillMap = delegate.mapLine(line,
7.             lineNumber);
8.         result.setAccountID((String)creditBillMap.get("accountID"));
9.         result.setName((String)creditBillMap.get("name"));
10.        result.setAmount((Double)creditBillMap.get("amount"));
11.        result.setDate((String)creditBillMap.get("date"));
12.        result.setAddress((String)creditBillMap.get("address"));
13.        return result;
14.    }
15.    省略 get, set 操作.....
16. }
```

其中，2 行：引用 `JsonLineMapper`，负责完成数据转换。

6 行：通过 `JsonLineMapper` 获取 `Map<String, Object>` 类型数据。

7~11 行：将 `Map<String, Object>` 结构数据转换为 `CreditBill` 对象。

JsonParser：Spring Batch 框架中依赖的外部组件，负责实现 JSON 格式数据与 Java 类型数据之间的转换。

JSON 格式文件读取配置，参见代码清单 6-35。

代码清单 6-35 配置读 JSON 格式文件

```
1. <bean:bean id="jsonItemReader"
2.     class="org.springframework.batch.item.file.FlatFileItemReader">
3.     <bean:property name="resource"
4.         value="classpath:ch06/data/flat/credit-card-bill-201303.
5.             json"/>
6.     <bean:property name="recordSeparatorPolicy"
7.         ref="jsonRecordSeparatorPolicy"/>
8.     <bean:property name="lineMapper" ref="wrappedJsonLineMapper"/>
9. </bean:bean>
10.
11. <bean:bean id="jsonRecordSeparatorPolicy"
12.     class="org.springframework.batch.item.file.separator.
13.         JsonRecordSeparatorPolicy"/>
14.
15. <bean:bean id="wrappedJsonLineMapper"
16.     class="com.juxtapose.example.ch06.flat.WrappedJsonLineMapper">
17.     <bean:property name="delegate" ref="jsonLineMapper"/>
18. </bean:bean>
```

```

15.     <bean:bean id="jsonLineMapper"
16.         class="org.springframework.batch.item.file.mapping.
            JsonLineMapper"/>

```

其中，5 行：声明使用 JSON 的记录分隔策略，使用 Bean 对象 jsonRecordSeparatorPolicy。

6 行：声明包装后的 JSON 数据转换器 WrappedJsonLineMapper，负责将 JSON 格式数据转换为 BillCredit 对象。

9~10 行：定义 JSON 格式的记录分割策略，使用类 JsonRecordSeparatorPolicy。

11~14 行：声明包装的行匹配策略 WrappedJsonLineMapper，数据转换由给 Spring Batch 框架提供的 jsonLineMapper 来表现。

15~16 行：声明 Spring Batch 框架自带的 JsonLineMapper，将 JSON 格式数据转换为 Map<String,Object>。

6.2.10 读记录跨多行文件

当 Flat 文件格式非标准时，通过实现记录分隔策略接口 RecordSeparatorPolicy 来实现非标准 Flat 格式文件。非标准的 Flat 文件格式有多种情况，例如记录跨多行、以特定的字符开头、以特定的字符结尾等。在代码清单 6-36 示例中每两行表示一条记录。

代码清单 6-36 记录跨多行（两行表示一条记录）

```

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08
2. ,Lu Jia Zui road
3. 4047390012345678,tom,320.00,2013-2-3 10:35:21
4. ,Lu Jia Zui road

```

默认的记录分隔策略 SimpleRecordSeparatorPolicy 或者 DefaultRecordSeparatorPolicy 已经不能处理此类文件。我们可以通过实现接口 RecordSeparatorPolicy，来自定义记录分隔策略 MultiLineRecordSeparatorPolicy。

读记录跨多行文件时，使用到的核心组件类图参见图 6-9。

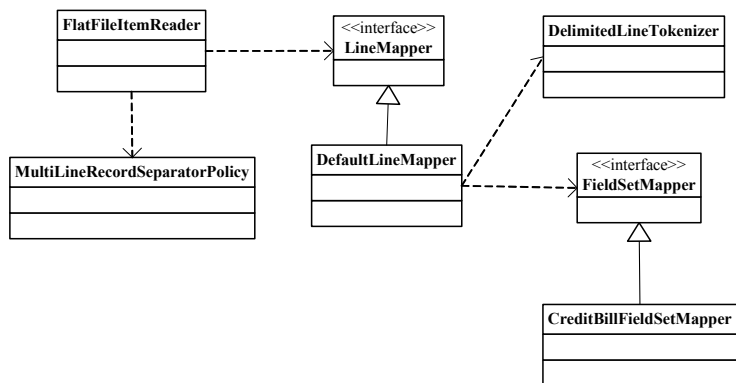


图 6-9 读记录跨多行文件时，使用到的核心组件类

在本类图中，除了 `MultiLineRecordSeparatorPolicy` 是自定义实现的，其他组件在前面章节已经详细介绍了，如有不明白的请参见前面的章节，此节不再赘述。

MultiLineRecordSeparatorPolicy：负责从文件中确认一条完整记录，在本实现中每读到指定的 4 个逗号分隔符，则认为是一条完整记录。

`MultiLineRecordSeparatorPolicy` 类定义参见代码清单 6-37。

代码清单 6-37 `MultiLineRecordSeparatorPolicy` 类定义

```
1. public class MultiLineRecordSeparatorPolicy implements
   RecordSeparatorPolicy {
2.     private String delimiter = ",";
3.     private int count = 0;
4.
5.     public boolean isEndOfRecord(String line) {
6.         return countDelimiter(line) == count;
7.     }
8.
9.     public String postProcess(String record) {
10.        return record;
11.    }
12.
13.    public String preProcess(String record) {
14.        return record;
15.    }
16.
17.    private int countDelimiter(String s) {
18.        String tmp = s;
19.        int index = -1;
20.        int count = 0;
21.        while ((index=tmp.indexOf(","))!=-1) {
22.            tmp = tmp.substring(index+1);
23.            count++;
24.        }
25.        return count;
26.    }
27. }
```

其中，2 行：定义读取的分隔符号。

3 行：分隔符总数，给定的字符串包含的分隔符个数等于此值，则认为是一条完整记录。

5~6 行：定义一条记录的完整规则。

17~25 行：统计给定内容包含分隔符的个数。

记录跨多行文件读取的配置，参见代码清单 6-38。

代码清单 6-38 配置读记录跨多行文件

```
1. <bean:bean id="multiLineItemReader"
2.     class="org.springframework.batch.item.file.FlatFileItemReader">
3.     <bean:property name="resource"
```

```

4.         value="classpath:ch06/data/flat/credit-card-bill-multiline-
           201303.csv"/>
5.     <bean:property name="recordSeparatorPolicy"
           ref="multiLineRecordSeparatorPolicy"/>
6.     <bean:property name="lineMapper" ref="lineMapper"/>
7. </bean:bean>
8.
9. <bean:bean id="multiLineRecordSeparatorPolicy"
10.    class="com.juxtapose.example.ch06.flat.
        MultiLineRecordSeparatorPolicy">
11.    <bean:property name="delimiter" value=", "/>
12.    <bean:property name="count" value="4"/>
13. </bean:bean>
14.
15. <bean:bean id="lineMapper"
16.    class="org.springframework.batch.item.file.mapping.
        DefaultLineMapper" >
17.    <bean:property name="lineTokenizer" ref="lineTokenizer" />
18.    <bean:property name="fieldSetMapper" ref="creditBillFieldSetMapper"/>
19. </bean:bean>
20.
21. <bean:bean id="lineTokenizer"
22.    class="org.springframework.batch.item.file.transform.
        DelimitedLineTokenizer">
23.    <bean:property name="delimiter" value=", "/>
24.    <bean:property name="names" value="accountID,name,amount,date,address"/>
25. </bean:bean>
26.
27. <bean:bean id="creditBillFieldSetMapper"
28.    class="com.juxtapose.example.ch06.flat.CreditBillFieldSetMapper">
29. </bean:bean>

```

其中,1~7行: 声明 FlatItemReader, 定义读取文件资源 credit-card-bill-multiline-201303.csv、行分隔策略、行转换策略。

9~13行: 声明多行的分隔策略, 分隔符为逗号, 4个分隔符为一条完整记录。

15~19行: 声明行转换策略, 将2行记录转换为 CreditBill 对象。

6.2.11 读混合记录文件

通常 Flat 文件中的记录格式是一致的, 在特殊情况下一个 Flat 文件中存在多种不同的记录格式, 通过特定的开头可以区分不同的记录。例如下面的文件, 以 40 开头的为信用卡消费记录, 以 30 开头的为借记卡消费记录情况。

混合记录文件格式参见代码清单 6-39。

代码清单 6-39 混合记录文件格式

```

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 3047390012345671,249.10,rose,2013/4/1 11:34:56
3. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
4. 3047390012345671,249.10,rose,2013/4/1 11:34:56
5. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
6. 3047390012345673,840.20,marry,2013/4/3 3:21:43
7. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road

```

Spring Batch 框架对文件中存在不同的记录格式同样有友好的支持,通过复杂数据转换类 `PatternMatchingCompositeLineMapper`, 可以为不同的记录定义不同的 `LineTokenizer` 和 `FieldSetMapper<T>` 来实现数据转换; 在执行过程中根据每条记录的内容找到匹配的 `LineTokenizer` 和 `FieldSetMapper<T>` 进行数据转换。

混合记录读的架构参见图 6-10。

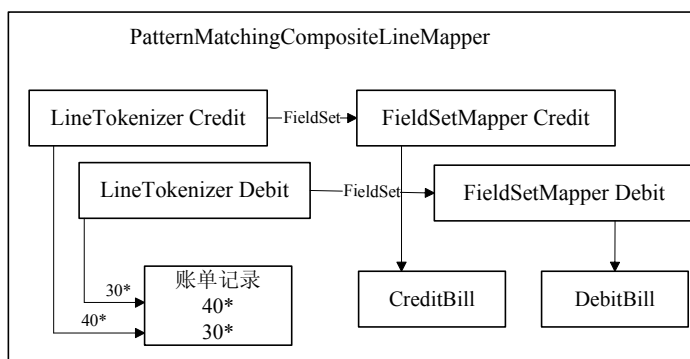


图 6-10 混合记录读的架构图

混合记录读主要类图参见图 6-11。

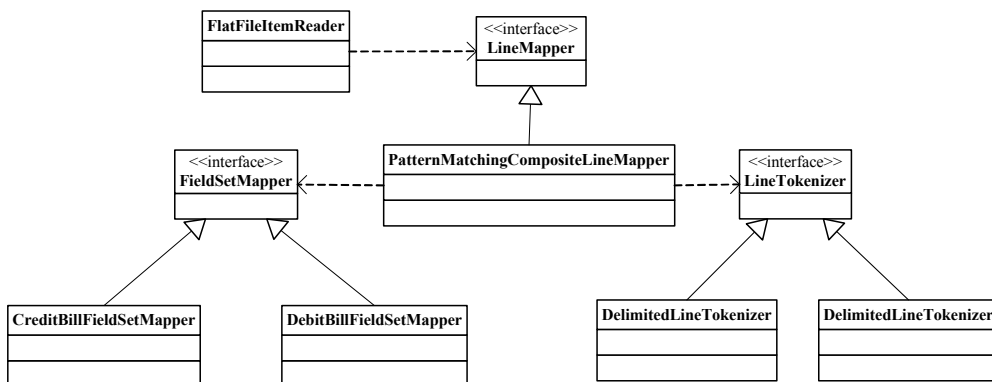


图 6-11 混合记录读主要类图

在图 6-11 中，PatternMatchingCompositeLineMapper：为不同的记录准备不同的 LineTokenizer 和 FieldSetMapper<T>来实现数据转换。

CreditBillFieldSetMapper：信用卡数据转换类，将 40* 格式的记录转换为领域对象 CreditBill。

DebitBillFieldSetMapper：借贷卡数据转换类，将 30* 格式的记录转换为领域对象 DebitBill。

DelimitedLineTokenizer：所有的记录是分隔符的，使用分隔符的字段分隔器来完成。

读混合记录文件配置参见代码清单 6-40。

代码清单 6-40 配置读混合记录文件

```
1.      <bean:bean id="heterogenousItemReader"
2.      class="org.springframework.batch.item.file.FlatFileItemReader">
3.          <bean:property name="resource"
4.              value="classpath:ch06/data/flat/credit-card-bill-
5.                  heterogenous-201303.csv"/>
6.          <bean:property name="lineMapper" ref="lineMapper"/>
7.      </bean:bean>
8.
9.      <bean:bean id="lineMapper" class="org.springframework.batch.item.
10.      file.mapping.PatternMatchingCompositeLineMapper">
11.          <bean:property name="tokenizers">
12.              <bean:map>
13.                  <bean:entry key="40*" value-ref="creditBillRecordTokenizer"/>
14.                  <bean:entry key="30*" value-ref="debitBillRecordTokenizer"/>
15.              </bean:map>
16.          </bean:property>
17.          <bean:property name="fieldSetMappers">
18.              <bean:map>
19.                  <bean:entry key="40*" value-ref="creditBillFieldSetMapper"/>
20.                  <bean:entry key="30*" value-ref="debitBillFieldSetMapper"/>
21.              </bean:map>
22.          </bean:property>
23.      </bean:bean>
```

其中，8~9 行：声明行转换策略 PatternMatchingCompositeLineMapper，有两个关键属性 tokenizers 和 fieldSetMappers；两个属性对应的类型分别是 Map<String, LineTokenizer>和 Map<String, FieldSetMapper<T>>。

10~15 行：声明属性 tokenizers，以 40 开头的通过 creditBillRecordTokenizer 处理记录，以 30 开头的通过 debitBillRecordTokenizer 处理记录。

16~21 行：声明属性 fieldSetMappers，以 40 开头的通过 creditBillFieldSetMapper 将记录转换为 CreditBill 对象，以 30 开头的通过 debitBillFieldSetMapper 将记录转换为 DebitBill 对象。

下面给出了不同记录的转换配置参见代码清单 6-41。

代码清单 6-41 配置不同记录的格式转换

```
1.      <bean:bean id="creditBillRecordTokenizer" parent="parentLineTokenizer">
2.          <bean:property name="names" value="accountID,name,amount,date,
3.              address" />
4.      </bean:bean>
5.      <bean:bean id="debitBillRecordTokenizer" parent="parentLineTokenizer">
6.          <bean:property name="names" value="accountID,amount,name,date" />
7.      </bean:bean>
8.      <bean:bean id="parentLineTokenizer" abstract="true"
9.          class="org.springframework.batch.item.file.transform.
10.              DelimitedLineTokenizer">
11.          <bean:property name="delimiter" value="," />
12.      </bean:bean>
13.      <bean:bean id="creditBillFieldSetMapper"
14.          class="com.juxtapose.example.ch06.flat.CreditBillFieldSetMapper">
15.      </bean:bean>
16.      <bean:bean id="debitBillFieldSetMapper"
17.          class="com.juxtapose.example.ch06.flat.DebitBillFieldSetMapper">
```

其中，1~10：定义不同记录转换为 FieldSet 对象。

12~18 行：定义不同记录将 FieldSet 对象转换为 CreditBill 或者 DebitBill 对象。

说明：混合记录文件之后可以将信用卡记录、借贷卡记录写入同一个文件也可以写入不同的文件中，上面例子中将两种类型的记录写入同一文件，如何写入不同的文件请参见写数据章节。

6.3 XML 格式文件

6.3.1 XML 解析

XM 是一种通用的数据交换格式，它的平台无关性、语言无关性、系统无关性，给数据集成与交互带来了极大的方便。XML 在 Java 领域的解析方式有两种，一种叫 SAX，另一种叫 DOM。SAX 是基于事件流的解析，DOM 是基于 XML 文档树结构的解析。

DOM 解析

DOM 解析器读入整个 XML 文档后构建一个驻留内存的树结构，然后代码就可以使用 DOM 接口来操作这个树结构。

优点：整个文档树在内存中，便于操作；支持删除、修改、重新排列等多种功能。

缺点：将整个文档调入内存（包括无用的节点），浪费时间和空间。

使用场合：一旦解析了文档还需多次访问这些数据。

SAX 解析

为解决 DOM 占用内存过大的问题，SAX 采用事件驱动的方式解析 XML 文档。当解析器发现元素开始、元素结束、文本、文档的开始或结束等时发送事件，开发者可以编写响应这些事件的代码，保存数据。

优点：不用事先调入整个文档，占用资源少。

缺点：不是持久的；事件过后，若没保存数据，那么数据就丢了；无状态性；从事件中只能得到文本，但不知该文本属于哪个元素。

使用场合：只需 XML 文档的少量内容，很少重复多次访问。

说明：DOM 解析和 SAX 解析都不适用于批处理的 XML 中的解析，因为两种处理方式都没有支持 Stream。下面我们将介绍一种支持 Stream 方式的 XML 解析方式。

StAX 解析

StAX（The Streaming API for XML），是一种利用拉模式解析（pull-parsing）XML 文档的 API。StAX 把重点放在流上，它提供了两套处理 XML 的 API：一种基于指针的 API，把 XML 文档当作一个标记（或事件）流来处理；允许应用程序检查解析器的状态，获得解析的上一个标记的信息，然后再处理下一个标记，依此类推；另一种较为高级的是基于迭代器的 API，把 XML 作为一系列事件对象来处理，每个对象和应用程序交换 XML 结构的一部分。应用程序根据需要定制解析事件的类型，然后将其转换成对应的具体类型，再然后利用定制事件提供的方法获得属于该事件的信息。

StAX 工作原理是通过一种基于事件迭代器（Iterator）的 API 让程序员去控制 xml 文档解析过程，程序遍历这个事件迭代器去处理每一个解析事件，解析事件可以看作是程序拉出来的，也就是程序促使解析器产生一个解析事件然后处理该事件，之后又促使解析器产生下一个解析事件，如此循环直到碰到文档结束符为止。

StAX 和 SAX 的区别如下。

SAX 也是基于事件处理的 XML 文档，但却是用推模式解析，解析器解析完整个 XML 文档后才产生解析事件，然后推给程序去处理这些事件。SAX 中解析器是工作主体，而事件处理器是由解析器驱动的，如果解析文档过程中产生问题，则剩余的所有文档就无法处理。

而 StAX 使用拉模式，解析器首先将 XML 文档所有的事件全部取出，然后通过处理程序处理这些事件。StAX 中处理器是工作主体，如果解析文档过程中产生问题，只会影响到出问题的部分，而其余部分处理不受影响。

6.3.2 Spring OXM

OXM 是 Object-to-XML-Mapping 的缩写，它是一个 O/X-mapper，负责将 Java 对象映射

为 XML 数据，或者将 XML 数据映射为 java 对象。

Spring framework 在 3.0 版本中引入了该特性，Spring O/X Mapper 仅定义由流行的第三方框架实现的统一的切面。要使用 Spring 的 O/X Mapper 功能，需提供一个能在 Java 对象和 XML 之间转换的组件。通常这样的组件有 Castor、XMLBeans、Java Architecture for XML Binding (JAXB)、JiBX 和 XStream 等。

Spring OXM 框架参见图 6-12。

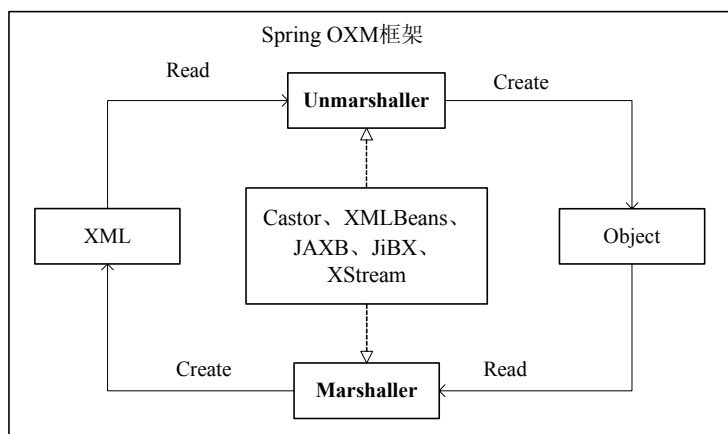


图 6-12 Spring OXM 框架图

图 6-12 中 OXM 框架通过 Unmarshaller 接口将 XML 文本反序列化为 Object 对象，通过接口 Marshaller 将 Object 对象序列化为 XML 文本。同时 Spring OXM 框架提供了内置的序列化和反序列化组件，包括 CastorMarshaller、JibxMarshaller、XmlBeansMarshaller、XStreamMarshaller、Jaxb2Marshaller 等。

编组、解组

编组或者称之为序列化（serializing），是指将 Java Bean 转换成 XML 文档的过程，这意味着将 Java Bean 中的所有字段和字段值都将作为 XML 元素或属性填充到 XML 文件中。

解组或者称之为反序列化（deserializing），是与编组完全相反的过程，即将 XML 文档转换为 Java Bean，这意味着 XML 文档的所有元素或属性都作为 Java 字段填充到 Java Bean 中。

Spring O/X Mapper 优点

- 易于配置

Spring 的 O/X Mapper 采用标准 Spring 框架的配置方式简化开发，Spring 的 Bean 库支持将实例化的 O/X 编组器注入那些使用编组器的对象。

- 一致的接口

Spring 的 O/X Mapper 框架统一提供两个接口：Marshaller 和 Unmarshaller，用于执行 O/X

功能。这些接口的实现完全对开发人员开放，开发人员可以轻松地切换它们而无须修改任何代码。例如，如果你一开始使用 Castor 进行 O/X 转换，但后来发现它缺乏你需要的某个功能，这时可以切换到 XStream 而无须任何代码更改（只需要做的是修改 Spring 配置文件以使用新的 O/X 框架）。

- 一致的异常层次结构

Spring 的 O/X Mapper 的另一个好处是统一的异常层次结构。Spring 框架将原始异常对象包装到 Spring 自身专为 O/X Mapper 建立的运行时异常（XMLMappingException）中。由于第三方提供商抛出的原始异常被包装到 Spring 运行时异常中，所以能够查明出现异常的根本原因。Spring 的 O/M Mapper 框架提供的异常包括 GenericMarshallingFailureException、ValidationFailureException、MarshallingFailureException、UnmarshallingFailureException

6.3.3 StaxEventItemReader

StaxEventItemReader 实现 ItemReader 接口，核心作用是将 XML 文件中的记录转换为 Java 对象。StaxEventItemReader 通过引用 OXM 组件完成对 XML 的读操作，负责将 XML 文件转换为 Java 对象，并交给处理或者写阶段。

StaxEventItemReader 结构关键属性

图 6-13 展示了 XML 文件读取的逻辑架构图，XMLEventReader 负责将文件按照 StaX 的模式读取指定的节点数据，然后交给反序列化组件 Unmarshaller 完成 Java 对象的生成。

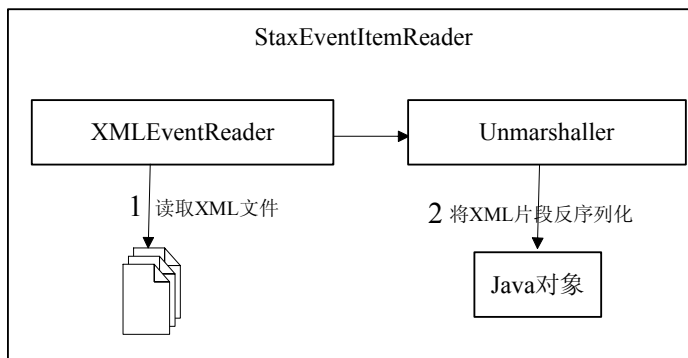


图 6-13 XML 文件读取的逻辑架构图

StaxEventItemReader 核心类架构图参见图 6-14。

StaxEventItemReader 中的核心类或者接口包括 Resource、Unmarshaller、XMLEventReader、FragmentEventReader、DefaultFragmentEventReader。Resource 表示需要处理的文件资源；Unmarshaller 负责将 XML 片段转换为 Java 对象，即进行反序列化；XMLEventReader 接口负责提供 StAX 方式对 XML 的解析；FragmentEventReader 接口继承接口 XMLEventReader，在

XMLEventReader 能力的基础上增加了将 XML 片段转换为 Document 能力，即增加了 StartDocument、EndDocument 两个事件；DefaultFragmentEventReader 是接口 FragmentEventReader 的默认实现，同时引用 XMLEventReader 对象，负责具体的 XML 处理工作。

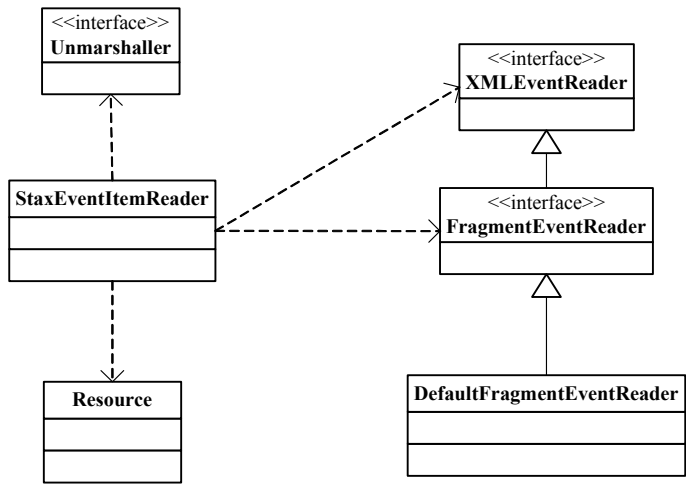


图 6-14 StaxEventItemReader 核心类图

在实际配置 StaxEventItemReader 时，只需要配置 Unmarshaller、Resource 两个属性即可。StaxEventItemReader 关键属性参见表 6-10。

表 6-10 StaxEventItemReader 关键属性

StaxEventItemReader 属性	类 型	说 明
fragmentRootElementName	String	需要转换为 Java 对象的根节点的名字
fragmentRootElementNameSpace	String	需要转换为 Java 对象的根节点的命名空间
resource	Resource	需要读取的资源文件
maxItemCount	String	能读取的最大条目数 默认值：Integer.MAX_VALUE
strick	Boolean	定义读取文件不存在时候的策略，如果为 true 则抛出异常， 如果为 false 表示不抛出异常。 默认值：true
unmarshaller	Int	Spring OXM 实现类，负责将 XML 内容转换为 Java 对象

配置 StaxEventItemReader

在给出详细配置文件之前，我们首先给出 XML 文件示例，参见代码清单 6-42，它通过 XML 的方式描述来信用卡的消费清单。

完整文件路径：ch06/data/xml/credit-card-bill-201303.xml。

代码清单 6-42 XML 示例文件

```
1.  <credits>
2.    <credit>
3.      <accountID>4047390012345678</accountID>
4.      <name>tom</name>
5.      <amount>100.00</amount>
6.      <date>2013-2-2 12:00:08</date>
7.      <address>Lu Jia Zui road</address>
8.    </credit>
9.    <credit>
10.     <accountID>4047390012345678</accountID>
11.     <name>tom</name>
12.     <amount>320.00</amount>
13.     <date>2013-2-3 10:35:21</date>
14.     <address>Lu Jia Zui road</address>
15.   </credit>
16.   .....
17. </credits>
```

信用卡账单文件的根节点为 `credits`，下面为具体的信用卡消费记录节点 `credit`，在根节点 `credits` 下面可以有多个信用卡消费记录 `credit`。

配置 `StaxEventItemReader` 参见代码清单 6-43。

完整的配置文件参见：`ch06/job/job-xml.xml`。

代码清单 6-43 配置 `StaxEventItemReader`

```
1.  <!-- XML 文件读取 -->
2.  <bean:bean id="xmlReader" scope="step"
3.    class="org.springframework.batch.item.xml.StaxEventItemReader">
4.    <bean:property name="fragmentRootElementName" value="credit"/>
5.    <bean:property name="unmarshaller" ref="creditMarshaller"/>
6.    <bean:property name="resource"
7.      value="classpath:ch06/data/xml/credit-card-bill-201303.xml"/>
8.  </bean:bean>
9.  <bean:bean id="creditMarshaller"
10.    class="org.springframework.xml.xstream.XStreamMarshaller">
11.    <bean:property name="aliases">
12.      <util:map id="aliases">
13.        <bean:entry key="credit"
14.          value="com.juxtapose.example.ch06.CreditBill"/>
15.      </util:map>
16.    </bean:property>
17.  </bean:bean>
18.  <bean:bean id="creditBill" scope="prototype"
19.    class="com.juxtapose.example.ch06.CreditBill"/>
```

其中，2~8 行：定义 XML 的读取类，主要属性包括 `fragmentRootElementName`、`unmarshaller`、`resource`；`fragmentRootElementName` 属性值为"credit"，表示读取 XML 文件中的节点为 credit 的元素转换为 Java 对象；`unmarshaller` 属性定义序列化组件；`resource` 属性定义读取的文件资源。

9~19 行：定义 XML 序列化的组件，此处使用 `XStream` 的方式进行序列化，使用 `Spring OXM` 提供的组件 `XStreamMarshaller` 进行序列化；只需要指定类别名 `aliases` 属性就可以将 XML 文件转换为特定的 Java 对象，此处使用 `com.juxtapose.example.ch06.CreditBill`，即将 `credits/credit` 路径下的 XML 文件内容转换为 `CreditBill` 对象。

6.4 读多文件

前面章节完整地介绍了对 Flat 格式、XML 格式文件的读取操作，细心的读者会发现上面所有的文件读取基本上是对单文件执行的。在实际的应用中，我们需要经常操作批量的文件，本节将为读者介绍如何对多文件的读取。以信用卡账单为例，在处理信用卡账单时，每月都会产生账单记录，在应用中期望一次处理多个账单文件：3、4、5 月份的信用卡账单，参见图 6-15。

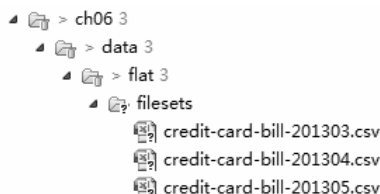


图 6-15 多文件处理示例

`Spring Batch` 框架提供了现有的组件 `MultiResourceItemReader` 支持对多文件的读取，通过 `MultiResourceItemReader` 读取批量文件非常简单。`MultiResourceItemReader` 通过代理的 `ItemReader` 来读取文件。

MultiResourceItemReader 结构关键属性

`MultiResourceItemReader` 组件实现 `ItemReader`、`ItemStream` 接口，并引用实现了接口 `ResourceAwareItemReaderItemStream` 的读组件，例如 `FlatFileItemReader`、`StaxEventItemReader`，通过这些具体的组件完成文件的读取。

`MultiResourceItemReader` 在处理文件的时候在一个线程中按照读取的文件顺序执行，在某些场景直接使用 `MultiResourceItemReader` 可能无法满足性能要求；如果需要高性能地处理海量数据可以通过 `Spring Batch` 框架提供的分区能力，并行计算来处理海量文件，分区请参照并行处理章节的介绍。

`MultiResourceItemReader` 核心类结构参见图 6-16。

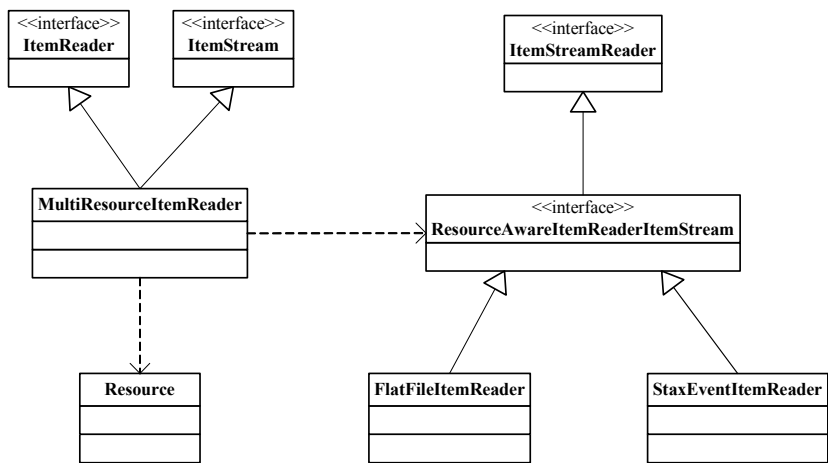


图 6-16 MultiResourceItemReader 核心类

MultiResourceItemReader 关键属性参见表 6-11。

表 6-11 MultiResourceItemReader 关键属性

MultiResourceItemReader 属性	类 型	说 明
delegate	ResourceAwareItemReaderItemStream	ItemReader 的代理，将 resources 中定义的文件代理给当前指定的 ItemReader 进行处理
resources	Resource[]	需要读取的资源文件列表
strict	boolean	定义读取文件不存在时候的策略，如果为 true 则抛出异常，如果为 false 表示不抛出异常。 默认值为 true
saveState	boolean	保存状态标识，读取资源时候是否保存当前读取的文件及当前文件是否读取条目记录的状态。 默认值为 true

配置 MultiResourceItemReader

在给出详细配置文件之前，我们首先给出多文件示例（参见代码清单 6-44、代码清单 6-45、代码清单 6-46），本例中期望将目录 ch06/data/flat/filesets/下面所有以“credit-card-bill-”开头，以“.csv”结尾的文件全部读取进行处理。下面给出了文件 credit-card-bill-201303.csv、credit-card-bill-201304.csv，credit-card-bill-201305.csv 的内容，每个文件均有 2 条消费记录；经过处理后，将记录汇集在一个目标文件中。

完整的文件路径：ch06/data/flat/filesets/credit-card-bill-201303.csv。

代码清单 6-44 示例文件 credit-card-bill-201303.csv

```

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
  
```

完整文件路径: ch06/data/flat/filesets/credit-card-bill-201304.csv。

代码清单 6-45 示例文件 credit-card-bill-201304.csv

```
1. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
2. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
```

完整文件路径: ch06/data/flat/filesets/credit-card-bill-201305.csv。

代码清单 6-46 示例文件 credit-card-bill-201305.csv

```
1. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
2. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road
```

配置 `MultiResourceItemReader` 参见代码清单 6-47。

完整配置文件参见: ch06/job/job-flatfile.xml。

代码清单 6-47 配置 `MultiResourceItemReader`

```
1.     <job id="fileSetsJob">
2.         <step id="fileSetsStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="multiResourceReader" writer="csvItemWriter"
5.                     commit-interval="2">
6.                     </chunk>
7.                 </tasklet>
8.             </step>
9.         </job>
10.
11.     <bean:bean id="multiResourceReader"
12.         class="org.springframework.batch.item.file.
13.             MultiResourceItemReader">
14.         <bean:property name="resources"
15.             value="classpath:/ch06/data/flat/filesets/
16.                 credit-card-bill-*.csv"/>
17.         <bean:property name="delegate" ref="flatFileItemReader"/>
18.     </bean:bean>
19.
20.     <bean:bean id="flatFileItemReader" scope="step"
21.         class="org.springframework.batch.item.file.FlatFileItemReader">
22.         <bean:property name="lineMapper" ref="lineMapper" />
23.         <bean:property name="strict" value="true"/>
24.     </bean:bean>
```

其中, 1~9 行: 定义多文件处理的作业, 读取多文件使用 `multiResourceReader`。

11~16 行: 定义多文件读取 `multiResourceReader`, 两个关键属性: `resources` 和 `delegate`; `resources` 属性表示需要读取的文件集合, 配置文件的前缀为“credit-card-bill-”, 后缀为“.csv”的文件; `delegate` 属性用于配置具体的文件读取 `ItemReader`, 本例中使用了 `FlatFileItemReader`

读取 Flat 格式的文件。

18~22 行：定义 flatFileItemReader，读取配置的 CSV 文件。

说明：MultiResourceItemReader 自身的实现不是线程安全的，不能在多线程中并发使用该组件。在一些场景中，文件数量多且文件内容较大，单线程处理不能满足时间或者性能上的要求，使用后面 11 章节介绍的分区功能，将数据分区后，可以使用多个线程甚至多个物理节点来并行处理数据。

6.5 读数据库

众多企业将数据存放在数据库中，友好的批处理框架需要友好的对数据库的读/写支持。Spring Batch 框架对读数据库提供了非常好的支持，包括基于 JDBC 和 ORM（Object-Relational Mapping）的读取方式；基于游标和分页的读取数据的 ItemReader 组件。

Spring Batch 框架提供的读数据库组件列表参见表 6-12。

表 6-12 Spring Batch 框架提供的读数据库组件

JdbcCursorItemReader	基于 JDBC 游标方式读数据库
HibernateCursorItemReader	基于 Hibernate 游标方式读数据库
StoredProcedureItemReader	基于存储过程读数据库
IbatisPagingItemReader	基于 Ibatis 分页读数据库
JpaPagingItemReader	基于 Jpa 方式分页读数据库
JdbcPagingItemReader	基于 JDBC 方式分页读数据库
HibernatePagingItemReader	基于 Hibernate 方式分页读取数据库

基于游标的读

在数据库中，游标是一个十分重要的概念。游标提供了一种对从表中检索出的数据进行操作的灵活手段，就本质而言，游标实际上是一种能从包括多条数据记录的结果集中每次提取一条记录的机制。

Spring Batch 框架提供了三种基于游标的读取方式，分别是：

- 基于 JDBC 的 JdbcCursorItemReader；
- 基于 Hibernate 实现的 HibernateCursorItemReader；
- 基于存储过程的 StoredProcedureItemReader。

基于分页的读

基于游标的数据库读取避免了一次查询大批量的数据导致消耗应用大量的内存，这是由于数据库提供了另外一种能力：基于分页的读，即通过分页读每次获取指定页大小的数据。Spring Batch 框架对分页读提供了默认的系统读组件。

Spring Batch 框架提供了四种基于分页的读取方式，既包括基于 JDBC 的分页读，也包含基于 ORM 的分页实现；四种分页读分别是：

- 基于 JDBC 的 `JdbcPagingItemReader`；
- 基于 Hibernate 实现的 `HibernatePagingItemReader`；
- 基于 iBatis 实现的 `IbatisPagingItemReader`；
- 基于 Jpa 实现的 `JdbcPagingItemReader`。

6.5.1 JdbcCursorItemReader

Spring Batch 框架提供了对 JDBC 读取支持的组件 `JdbcCursorItemReader`。`JdbcCursorItemReader` 实现 `ItemReader` 接口，核心作用是将数据库中的记录转换为 Java 对象。`JdbcCursorItemReader` 通过引用 `PreparedStatement`、`RowMapper`、`PreparedStatementSetter` 关键接口实现上面的目的；在 `JdbcCursorItemReader` 将数据库记录转换为 Java 对象时主要有两步工作：首先根据 `PreparedStatement` 从数据库中获取结果集 `ResultSet`；其次使用 `RowMapper` 将结果集 `ResultSet` 转换为 Java 对象，具体步骤见图 6-17。

`JdbcCursorItemReader` 结构及关键属性

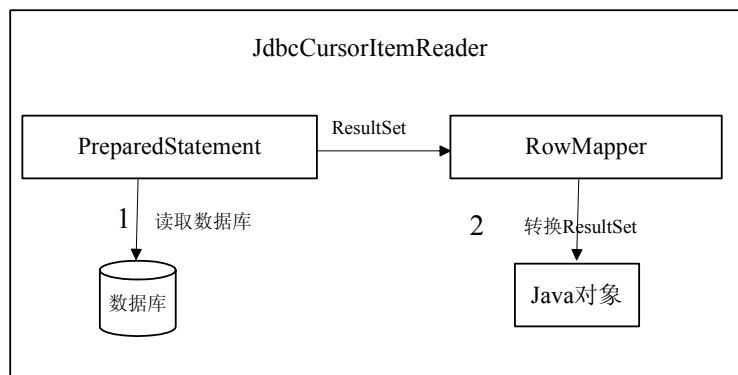


图 6-17 `JdbcCursorItemReader` 核心操作步骤

`JdbcCursorItemReader` 与关键接口 `PreparedStatement`、`PreparedStatementSetter`、`RowMapper` 之间的类图参见图 6-18。

`JdbcCursorItemReader` 引用 `javax.sql.DataSource`，`DataSource` 提供读取的数据库信息；`JdbcCursorItemReader` 通过 `PreparedStatement` 对数据库进行读/写，返回结果集 `ResultSet`；对象 `PreparedStatementSetter` 为读取的 SQL 提供参数设置能力；`RowMapper` 负责将 `ResultSet` 对象转换为 Java 对象。

关键接口、类说明参见表 6-13。

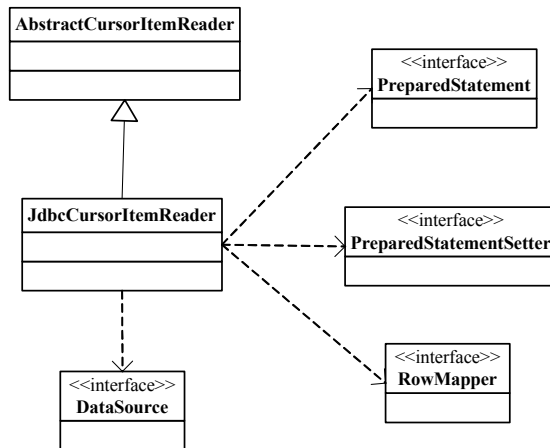


图 6-18 JdbcCursorItemReader 类关系图

表 6-13 关键接口、类说明

关 键 类	说 明
DataSource	提供读取数据库的数据源信息
PreparedStatement	PreparedStatement 实例包含已编译的 SQL 语句，并提供执行 SQL 语句的能力，返回结果集 ResultSet
PreparedStatementSetter	负责为 PreparedStatement 中的 SQL 提供参数支持
RowMapper	负责将结果集 ResultSet 转换为 Java 对象

JdbcCursorItemReader 关键属性参见表 6-14。

表 6-14 JdbcCursorItemReader 关键属性

JdbcCursorItemReader 属性	类 型	说 明
dataSource	DataSource	数据源，通过该属性指定使用的数据库信息
driverSupportsAbsolute	Boolean	数据库驱动是否支持结果集的绝对定位。 默认值：false
fetchSize	int	设置 ResultSet 每次向数据库取的行数； setFetchSize 的意思是当调用 rs.next 时，ResultSet 会一次性从服务器上取多少行数据回来，这样在下次 rs.next 时，可以直接从内存中获取数据而不需要网络交互，提高了效率。 默认值：-1
ignoreWarnings	Boolean	是否忽略 SQL 执行期间的警告信息，true 表示忽略警告，false 表示会抛出异常。 默认值：true

续表

JdbcCursorItemReader 属性	类 型	说 明
maxRows	int	设置结果集做大行数。 默认值: -1 (表示 unlimited)
preparedStatementSetter	PreparedStatementSetter	SQL 语句参数准备, 可以使用批处理框架提供的 ListPreparedStatementSetter, 也可以自定义实现该接口
queryTimeout	int	查询超时时间。如果超时, 将抛出超时异常。 默认值: -1 (表示永超时)
rowMapper	RowMapper	将结果集 ResultSet 转换为指定的 Pojo 对象类; 需要实现 RowMapper 接口; 默认可以使用 BeanPropertyRowMapper
saveState	Boolean	是否将当前 Reader 的状态保存到 job repository 中, 即当前读取到数据库的行数; 在操作 ItemStream#update()中被持久化。 默认值: true
sql	String	需要执行的 SQL
useSharedExtendedConnection	Boolean	不同游标间是否共享数据库连接; 如果共享则必须在同一个事务当中, 否则使用不同的事务。 默认值: false
verifyCursorPosition	Boolean	处理完当前行后, 是否校验游标位置。 默认值: true

配置 JdbcCursorItemReader

使用 JdbcCursorItemReader 至少需要配置 dataSource、sql、rowMapper 三个属性; dataSource 指定访问的数据源, sql 用于指定查询的 SQL 语句, rowMapper 用于将结果集 ResultSet 转换为 Java 业务对象。

在给出详细的配置文件之前, 我们首先准备 SQL (使用的为 MySQL 数据库) 脚本, 示例中使用信用卡账单表, 存放信用卡消费记录情况, 主要字段包括 ID、ACCOUNTID、NAME、AMOUNT、DATE、ADDRESS; 建表脚本参见代码清单 6-48。

完整内容参见文件: ch06/db/create-tables-mysql.sql。

代码清单 6-48 建表脚本 create-tables-mysql.sql

```

1. CREATE TABLE t_credit
2.     (ID VARCHAR(10),
3.      ACCOUNTID VARCHAR(20),
4.      NAME VARCHAR(10),
5.      AMOUNT NUMERIC(10,2),

```

```

6.         DATE VARCHAR(20),
7.         ADDRESS VARCHAR(128),
8.         primary key (ID)
9.     )
10.    ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

配置 `JdbcCursorItemReader` 参见代码清单 6-49。

完整配置文件参见：ch06/job/job-db-jdbc.xml。

代码清单 6-49 配置 `JdbcCursorItemReader`

```

1. <bean:bean id="jdbcItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
        JdbcCursorItemReader" >
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
5.     DATE,ADDRESS from t_credit where id between 1 and 5 "/>
6.     <bean:property name="rowMapper">
7.         <bean:bean class="org.springframework.jdbc.core.
            BeanPropertyRowMapper">
8.             <bean:property name="mappedClass"
9.                 value="com.juxtapose.example.ch06.CreditBill"/>
10.        </bean:bean>
11.    </bean:property>
12. </bean:bean>

```

其中，3 行：配置访问的数据源。

4 行：定义需要查询的 SQL 语句，查询 `t_credit` 表，且 `id` 在 1~5 范围内的记录。

6~10 行：指定 `rowMapper` 属性，将结果集 `ResultSet` 转换为 `CreditBill` 对象；此时使用 Spring 提供的 `BeanPropertyRowMapper` 完成，通过设置属性 `mappedClass`，自动将结果集 `ResultSet` 和 `CreditBill` 对象映射（即将表中的字段名称映射到 `CreditBill` 对象的属性中）。

配置数据源

数据库访问需要数据源定义，在 Spring Batch 框架中数据源的配置是标准的 Spring 配置。具体的配置信息参见代码清单 6-50。

代码清单 6-50 配置数据源

```

1. <context:property-placeholder location="classpath:/ch06/properties/
    batch-mysql.properties" />
2. <bean:bean id="dataSource"
3.     class="org.springframework.jdbc.datasource.
        DriverManagerDataSource">
4.     <bean:property name="driverClassName">
5.         <bean:value>${datasource.driver}</bean:value>

```

```

6.         </bean:property>
7.         <bean:property name="url">
8.             <bean:value>${datasource.url}</bean:value>
9.         </bean:property>
10.        <bean:property name="username" value="${datasource.username}">
11.            </bean:property>
12.        <bean:property name="password" value="${datasource.password}">
13.            </bean:property>
14.    </bean:bean>

```

其中，1 行：指定加载的属性配置文件，方便动态指定数据源的属性。

2~12 行：配置数据源的具体实现，分别指定属性 `driverClassName`、`url`、`username`、`password`。

自定义 RowMapper

使用 `RowMapper` 进行数据映射时，可以使用 Spring 框架提供的 `BeanPropertyRowMapper` 来自动将数据库的字段映射到指定的 Java Bean 上；另外通过实现接口 `org.springframework.jdbc.core.RowMapper<T>`，可以自定义 `RowMapper` 的实现类。`RowMapper` 接口定义参见代码清单 6-51，根据需要完成自定义的数据转换功能。

代码清单 6-51 `RowMapper` 接口定义

```

1. public interface RowMapper<T> {
2.     T mapRow(ResultSet rs, int rowNum) throws SQLException;
3. }

```

其中，2 行：核心转换方法，参数为结果集和当前游标所在的位置。

本例中使用自定义的信用卡账单转换类 `com.juxtapose.example.ch06.db.CreditBillRowMapper`。`CreditBillRowMapper` 的代码实现参见代码清单 6-52。

代码清单 6-52 自定义 `RowMapper` 实现 `CreditBillRowMapper`

```

1. public class CreditBillRowMapper implements RowMapper<CreditBill> {
2.
3.     public CreditBill mapRow(ResultSet rs, int rowNum) throws SQLException {
4.         CreditBill bill = new CreditBill();
5.         bill.setAccountID(rs.getString("ACCOUNTID"));
6.         bill.setAddress(rs.getString("ADDRESS"));
7.         bill.setAmount(rs.getDouble("AMOUNT"));
8.         bill.setDate(rs.getString("DATE"));
9.         bill.setName(rs.getString("NAME"));
10.        return bill;
11.    }
12. }

```

其中，3~11 行：业务实现代码，将给定的结果集 `ResultSet` 转化为 `CreditBill` 对象。

接下来我们使用自定义的 `CreditBillRowMapper` 配置 `JdbcCursorItemReader`，参见代码清

单 6-53。CreditBillRowMapper 将数据库中的行记录转换为领域对象。

代码清单 6-53 配置 JdbcCursorItemReader

```
1. <bean:bean id="jdbcItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
        JdbcCursorItemReader" >
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
5.     DATE,ADDRESS from t_credit where id between 1 and 5 "/>
6.     <bean:property name="rowMapper" ref="custCreditRowMapper" />
7. </bean:bean>
8. <bean:bean id="custCreditRowMapper"
9.     class="com.juxtapose.example.ch06.db.CreditBillRowMapper"/>
```

其中, 6 行: 使用自定义的 custCreditRowMapper 来指定 rowMapper 属性, 负责数据转换, 将结果集 ResultSet 转换为 CreditBill 对象。

8~9 行: 声明实现 RowMapper 接口的类 CreditBillRowMapper。

SQL 参数绑定

当 SQL 语句执行过程中需要动态指定参数时, 可以使用属性 preparedStatementSetter 来设置, 该属性是类型为 PreparedStatementSetter 的对象。

Spring Batch 框架提供了 PreparedStatementSetter 的默认实现 ListPreparedStatementSetter (org.springframework.batch.core.resource.ListPreparedStatementSetter), 根据给定的参数顺序设置 SQL 语句的参数。

使用 preparedStatementSetter 属性配置 JdbcCursorItemReader, 参见代码清单 6-54。

代码清单 6-54 使用 preparedStatementSetter 配置 JdbcCursorItemReader

```
1. <bean:bean id="jdbcParameterItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
        JdbcCursorItemReader" >
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
5.     DATE,ADDRESS from t_credit where id between 1 and ? "/>
6.     <bean:property name="rowMapper" ref="custCreditRowMapper" />
7.     <bean:property name="preparedStatementSetter" ref="paramStatement
        Setter"/>
8. </bean:bean>
9.
10. <bean:bean id="paramStatementSetter" scope="step"
11.     class="org.springframework.batch.core.resource.
        ListPreparedStatementSetter">
12.     <bean:property name="parameters">
```

```

13.         <bean:list>
14.             <bean:value>#{jobParameters['id']}</bean:value>
15.         </bean:list>
16.     </bean:property>
17. </bean:bean>

```

其中，4~5 行：注意 SQL 语句变化，此处查找范围是 1 到待定的范围，待定的范围通过后续的属性 `preparedStatementSetter` 设置。

10~16 行：使用 `ListPreparedStatementSetter` 来设置 SQL 语句需要的参数。

14 行：使用参数后绑定技术来实现，此处使用执行 Job 时候传入的参数“id”作为 SQL 语句查询的范围。

使用代码清单 6-55 执行定义的 `dbReadJob`。

完整代码参见：`test.com.juxtapose.example.ch06.JobLaunchJDBC`。

代码清单 6-55 执行 `dbReadJob`

```

1. executeJob("ch06/job/job-db-jdbc.xml", "dbReadJob",
2.     new JobParametersBuilder().addDate("date", new Date()).
        addString("id", "5"));

```

其中，2 行：传入参数 `id=5`，最终 SQL 执行查询的脚本为“`select ID,ACCOUNTID,NAME,AMOUNT,DATE, ADDRESS from t_credit where id between 1 and 5`”。

自定义 `PreparedStatementSetter`

上面使用了 Spring 框架自带的 `ListPreparedStatementSetter`，开发者可以根据自己业务需求实现自定义的 `PreparedStatementSetter`，进行参数设置。`PreparedStatementSetter` 接口定义参见代码清单 6-56。

代码清单 6-56 `PreparedStatementSetter` 接口定义

```

1. public interface PreparedStatementSetter {
2.     void setValues(PreparedStatement ps) throws SQLException;
3. }

```

其中，2 行：核心方法，为参数 `PreparedStatement` 设置 SQL 需要的参数。

自定义 `CreditBillPreparedStatementSetter` 类的实现参见代码清单 6-57。

完整代码参见：`com.juxtapose.example.ch06.db.CreditBillPreparedStatementSetter`。

代码清单 6-57 `CreditBillPreparedStatementSetter` 类定义

```

1. public class CreditBillPreparedStatementSetter implements
2.     PreparedStatementSetter {
3.     public void setValues(PreparedStatement ps) throws SQLException {
4.         ps.setString(1, "5");
5.     }
6. }

```

其中 3~5 行：完成参数设置，此处设置第一个参数为“5”。

使用自定义的 `CreditBillPreparedStatementSetter` 配置 `JdbcCursorItemReader`，参见代码清单 6-58。

代码清单 6-58 自定义参数配置 `JdbcCursorItemReader`

```
1. <bean:bean id="jdbcParameterItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
        JdbcCursorItemReader" >
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
5.     DATE,ADDRESS from t_credit where id between 1 and ? "/>
6.     <bean:property name="rowMapper" ref="custCreditRowMapper" />
7.     <bean:property name="preparedStatementSetter"
8.         ref=" custPreparedStatementSetter "/>
9. </bean:bean>
10.
11. <bean:bean id="custPreparedStatementSetter"
12.     class="com.juxtapose.example.ch06.db.
        CreditBillPreparedStatementSetter"/>
13.
```

其中，7~8 行：使用自定义的 `custPreparedStatementSetter` 设置属性 `preparedStatementSetter`。

11~13 行：声明 `custPreparedStatementSetter`，实现类为 `com.juxtapose.example.ch06.db.CreditBillPreparedStatementSetter`。

6.5.2 HibernateCursorItemReader

对象关系映射（Object Relational Mapping，ORM）是一种为解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，使得 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。Spring Batch 框架对 ORM 类型的 Hibernate 提供了基于游标的读 `ItemReader`。

`HibernateCursorItemReader` 实现 `ItemReader` 接口，核心作用是将数据库中的记录通过 ORM 的方式转换为 Java Bean 对象。学会使用基于 JDBC 的读数据库后，使用 `HibernateCursorItemReader` 非常简单。下面我们先看 `HibernateCursorItemReader` 的关键支持的属性，参见表 6-15。

HibernateCursorItemReader 结构及关键属性

表 6-15 HibernateCursorItemReader 关键属性

HibernateCursorItemReader 属性	类 型	说 明
fetchSize	int	设置 ResultSet 每次向数据库取的行数；setFetchSize 的意思是当调用 rs.next 时，ResultSet 会一次性从服务器上取多少行数据回来，这样在下次 rs.next 时，可以直接从内存中获取数据而不需要网络交互，提高了效率。 默认值：-1
maxItemCount	int	设置结果集做大行数。 默认值：Integer.MAX_VALUE
parameterValues	String	Statement 的参数值
queryProvider	HibernateQueryProvider	生成 HQL 的查询类
queryString	String	HQL 查询语句
sessionFactory	SessionFactory	Hibernate 的 SessionFactory，负责与数据库进行交互
useStatelessSession	Boolean	是否使用无状态的会话。 默认值：true

配置 HibernateCursorItemReader

使用 HibernateCursorItemReader 至少需要配置 sessionFactory 和 queryString 两个属性；sessionFactory 是 Hibernate 的会话工厂类，用于与数据库进行交互访问；queryString 用于指定查询的 HQL 语句。

在给出详细配置文件之前，我们首先准备实体对象（参见代码清单 6-59）、配置 Hibernate 的.cfg.xml（参见代码清单 6-60）。本节示例仍然使用 6.5.1 章节使用的数据库脚本。

配置数据实体对象

完整内容参见类 com.juxtapose.example.ch06.hibernate.CreditBill。

代码清单 6-59 数据实体对象 CreditBill

```
1. @Entity
2. @Table(name="t_credit")
3. public class CreditBill {
4.     @Id
5.     @Column(name="ID")
6.     private String id;
7.
8.     @Column(name="ACCOUNTID")
```

```

9.     private String accountID = "";    /** 银行卡账户 ID */
10.
11.     @Column(name="NAME")
12.     private String name = "";        /** 持卡人姓名 */
13.
14.     @Column(name="AMOUNT")
15.     private double amount = 0;       /** 消费金额 */
16.
17.     @Column(name="DATE")
18.     private String date;              /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */
19.
20.     @Column(name="ADDRESS")
21.     private String address;           /** 消费场所 */
22.
23.     .....
24. }

```

其中，1 行：@Entity 注释声明该类为持久类，将一个 JavaBean 类声明为一个实体的数据库表映射类，建议实现序列化。默认情况下，所有的类属性都为映射到数据表的持久性字段。若在类中，添加另外属性，而非映射自数据库的，要用 Transient 来注解。

2 行：@Table(name="t_credit") 持久性映射的表，表名为“t_credit”。@Table 是类一级的注解，定义在@Entity 之下，为实体 Bean 映射表、目录和 schema 的名字，默认为实体 Bean 的类名，不包含包名。

4 行：@Id，用于标识数据表的主键。

5 行：@Column(name="ID") 表示属性对应的数据库列的字段名。

配置 hibernate 的 cfg 文件

数据实体配置完成后，需要配置 hibernate 的配置文件，用于声明上面的数据实体。

完整文件参见：ch06/cfg/hibernate.cfg.xml。

代码清单 6-60 配置 hibernate 的 cfg 文件

```

1. <hibernate-configuration>
2.     <session-factory>
3.         <mapping class="com.juxtapose.example.ch06.hibernate.CreditBill"/>
4.     </session-factory>
5. </hibernate-configuration>

```

其中，3 行：声明 Hibernate 中使用的数据实体类 CreditBill。

配置 HibernateCursorItemReader

完整配置文件参见：ch06/job/job-db-hibernate.xml。

代码清单 6-61 展示了从数据库表 t_credit 中读取数据。

代码清单 6-61 配置 HibernateCursorItemReader

```
1.      <bean:bean id="hibernateItemReader" scope="step"
2.          class="org.springframework.batch.item.database.Hibernate
3.          CursorItemReader" >
4.          <bean:property name="sessionFactory" ref="sessionFactory"/>
5.          <bean:property name="queryString"
6.              value="from CreditBill where id between :begin and :end "/>
7.          <bean:property name="parameterValues">
8.              <bean:map>
9.                  <bean:entry key="begin" value="#{jobParameters['begin']}" />
10.                 <bean:entry key="end" value="#{jobParameters['end']}" />
11.             </bean:map>
12.         </bean:property>
13.     </bean:bean>
14.
15.     <bean:bean id="sessionFactory"
16.         class="org.springframework.orm.hibernate3.
17.         LocalSessionFactoryBean">
18.         <bean:property name="dataSource" ref="dataSource"/>
19.         <bean:property name="configurationClass"
20.             value="org.hibernate.cfg.AnnotationConfiguration"/>
21.         <bean:property name="configLocation"
22.             value="classpath:/ch06/cfg/hibernate.cfg.xml"/>
23.         <bean:property name="hibernateProperties">
24.             <bean:value>
25.                 hibernate.dialect=org.hibernate.dialect.MySQLDialect
26.                 hibernate.show_sql=true
27.             </bean:value>
28.         </bean:property>
29.     </bean:bean>
```

其中，3 行：属性 sessionFactory，定义 Hibernate 的会话访问工厂类，用于 ORM 映射，访问数据库表信息。

4~5 行：属性 queryString 定义需要查询的 HQL 语句，“from CreditBill where id between :begin and :end”，该 HQL 语句有两个查询参数，分别是:begin 和:end；需要通过属性 parameterValues 指定参数值。

6~11 行：属性 parameterValues 用于指定属性 queryString 中的变量:begin 和:end；此处使用了参数后绑定技术，在 Job 执行时候可以通过指定 JobParameter 给 begin 和 end 参数赋值。

14~27 行：声明 hibernate 使用的会话工厂，使用 hibernate 提供的 LocalSessionFactoryBean，需要为下面的属性赋值 dataSource（数据源）、configurationClass（通过注解的方式获取）、configLocation（配置文件地址）和 hibernateProperties（基本属性信息）。

使用代码清单 6-62 执行定义的 `hibernateReadJob`。完整代码参见：`test.com.juxtapose.example.ch06.JobLaunchHibernate`。

代码清单 6-62 执行 `hibernateReadJob`

```
1. executeJob("ch06/job/job-db-hibernate.xml", "hibernateReadJob",
2.           new JobParametersBuilder().addDate("date", new Date())
3.           .addString("begin", "1").addString("end", "4"));
```

其中，3 行：传入参数 `begin=1`, `end=4`；最终 SQL 执行查询的脚本为“`from CreditBill where id between 1 and 4`”。

6.5.3 StoredProcedureItemReader

Spring Batch 框架对存储过程提供了支持，`StoredProcedureItemReader` 提供了对存储过程的支持，其运行和 `JdbcCursorItemReader` 类似，均是获取游标对象，然后转换为 `JavaBean` 对象。存储过程获取游标通常有如下几种方式：

- (1) 执行存储过程直接返回结果集 `ResultSet`，数据库 `SQL Server`、`Sybase`、`DB2`、`Derby`、`MySQL` 中的存储过程支持直接返回结果集 `ResultSet`；
- (2) 通过返回 `Out` 类型的参数，获取引用类型的游标，数据库 `Oracle`、`PostgreSQL` 提供此类型的支持能力；
- (3) 通过 `function` 调用返回结果集 `ResultSet`。

StoredProcedureItemReader 关键属性

`StoredProcedureItemReader` 中可以使用 `JdbcCursorItemReader` 中定义的所有属性，同时 `StoredProcedureItemReader` 支持下面特有的属性，参见表 6-16 `StoredProcedureItemReader` 支持的特有属性。

表 6-16 `StoredProcedureItemReader` 支持的特有属性

StoredProcedureItemReader 属性	类 型	说 明
<code>function</code>	<code>Boolean</code>	是否调用存储过程的 <code>function</code> 。 默认值： <code>false</code>
<code>parameters</code>	<code>SqlParameter</code>	存储过程的参数类型
<code>procedureName</code>	<code>String</code>	调用的存储过程名称
<code>refCursorPosition</code>	<code>int</code>	使用 <code>OUT</code> 类型参数时候，指定 <code>OUT</code> 类型参数在参数列表中的位置， <code>index</code> 的列表从 0 开始。 默认值：0

配置 `StoredProcedureItemReader`

使用 `StoredProcedureItemReader` 与如何从存储过程获取游标有较大的关系，下面的示例

使用上面的第一种，即执行存储过程直接返回结果集 `ResultSet`，本示例使用 `MySQL` 数据库。

在给出详细配置文件之前，我们首先准备存储过程（代码清单 6-63）；本节示例仍然使用 6.5.1 章节使用的数据库表 `t_credit`。

创建存储过程完整内容参见：ch06/db/create-stored-procedure-mysql.sql。

代码清单 6-63 创建存储过程

```
1. DROP PROCEDURE IF EXISTS query_credit;
2. CREATE PROCEDURE query_credit() SELECT * FROM t_credit;
```

其中，1 行：删除已经存在的存储过程。

2 行：创建存储过程，本存储过程查询 `t_credit` 中所有的信息。

1. 存储过程直接返回 `ResultSet`

完整配置文件参见：ch06/job/job-db-stored-procedure.xml。

代码清单 6-64 展示了从存储过程 “`query_credit`” 中读取数据。

代码清单 6-64 配置 `StoredProcedureItemReader`

```
1. <bean:bean id="storedProcedureItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
        StoredProcedureItemReader" >
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="procedureName" value="query_credit"/>
5.     <bean:property name="rowMapper">
6.         <bean:bean class="org.springframework.jdbc.core.
            BeanPropertyRowMapper">
7.             <bean:property name="mappedClass"
8.                 value="com.juxtapose.example.ch06.CreditBill"/>
9.         </bean:bean>
10.    </bean:property>
11. </bean:bean>
```

其中，3 行：属性 `dataSource` 指定访问的数据源。

4 行：属性 `procedureName` 声明需要执行的存储过程的名称。

5~10 行：属性 `rowMapper` 定义如何将结果集 `ResultSet` 转换为 `JavaBean` 对象，本示例使用 `BeanPropertyRowMapper` 将 `ResultSet` 根据属性名自动映射到 `CreditBill` 对象。

读者可能注意到，该存储过程的调用和基于 `JDBC` 的调用类似，唯一不同的是在存储过程中使用属性 `procedureName` 指定需要调用的存储过程；而 `JDBC` 中使用属性 `sql` 属性指定需要调用的 `SQL` 语句。

2. 使用 `function`

如果使用 `function`，需要将属性 `"function"` 设置为 `true`，代码清单 6-65 的示例代码给出了

如何使用数据库的 function 功能。

代码清单 6-65 使用 function 配置 StoredProcedureItemReader

```
1. <bean:bean id="storedProcedureItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
        StoredProcedureItemReader" >
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="procedureName" value="query_credit"/>
5.     < bean:property name="function" value="true"/>
6.     <bean:property name="rowMapper">
7.         .....
8.     </bean:property>
9. </bean:bean>
```

其中，5 行：属性 function 设置为 true，表示使用数据库的 function 获取数据。

3. 通过 Out 参数返回 ResultSet

使用 Out 参数返回结果集，需要额外使用 2 个属性：parameters 与 refCursorPosition。属性 refCursorPosition 用于指定 Out 参数在参数列表中的位置；属性 parameters 指定使用的参数。代码清单 6-66 给出了如何使用 Out 参数返回结果集。

代码清单 6-66 通过 Out 参数获取结果集

```
1. <bean:bean id="storedProcedureItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
        StoredProcedureItemReader" >
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="procedureName" value="query_credit"/>
5.     <bean:property name="refCursorPosition" value="0"/>
6.     <bean:property name="parameters">
7.         <bean:list>
8.             <bean:bean class="org.springframework.jdbc.core.
                SqlOutParameter">
9.                 <constructor-arg index="0" value="products"/>
10.                <constructor-arg index="1">
11.                    <util:constant static-field="oracle.jdbc.
                        OracleTypes.CURSOR"/>
12.                </constructor-arg>
13.            </bean:bean>
14.        </bean:list>
15.    </bean:property>
16.    <bean:property name="rowMapper">
17.        .....
18.    </bean:property>
19. </bean:bean>
```

其中，5 行：属性 `refCursorPosition` 指定 Out 参数在参数列表中的位置。
6~15 行：属性 `parameters` 定义存储过程的参数信息。

6.5.4 JdbcPagingItemReader

Spring Batch 框架提供了对 JDBC 分页读取支持的组件 `JdbcPagingItemReader`。`JdbcPagingItemReader` 实现 `ItemReader` 接口，核心作用是将数据库中的记录通过分页的方式转换为 Java 对象。`JdbcPagingItemReader` 通过引用 `PagingQueryProvider`、`SimpleJdbcTemplate`、`RowMapper` 关键接口实现上面功能；在 `JdbcPagingItemReader` 将数据库记录转换为 Java 对象时主要有两步工作：首先根据 `SimpleJdbcTemplate` 与 `PagingQueryProvider` 从数据库中根据分页的大小获取结果集 `ResultSet`；其次使用 `RowMapper` 将结果集 `ResultSet` 转换为 Java 对象，具体步骤见图 6-19。

JdbcPagingItemReader 结构及关键属性

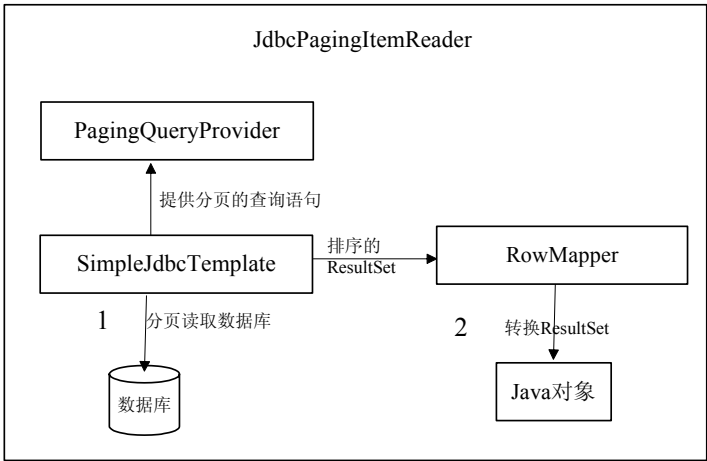


图 6-19 JdbcPagingItemReader 核心操作步骤

`JdbcPagingItemReader` 与关键接口 `SimpleJdbcTemplate`、`PagingQueryProvider`、`RowMapper` 之间的类图参见图 6-20。

`JdbcPagingItemReader` 引用 `javax.sql.DataSource`，`DataSource` 提供读取的数据库信息；`JdbcCursorItemReader` 通过 `SimpleJdbcTemplate` 对数据库进行读、写，使用 `PagingQueryProvider` 提供每次分页的查询语句，返回排序的结果集 `ResultSet`；如果需要为查询的 SQL 设置参数，可以通过属性 `parameterValues` 来设置参数值；`RowMapper` 负责将 `ResultSet` 对象转换为 Java 对象。`JdbcPagingItemReader` 关键接口、类说明参见表 6-17，`JdbcPagingItemReader` 关键属性参见表 6-18。

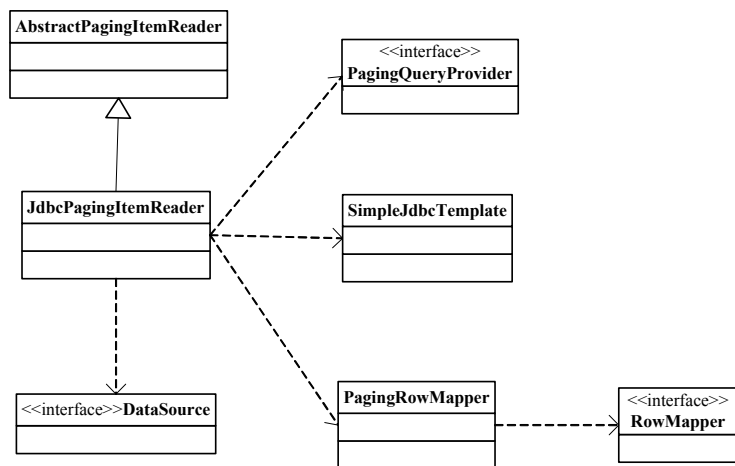


图 6-20 JdbcPagingItemReader 类关系图

表 6-17 JdbcPagingItemReader 关键接口、类说明

关 键 类	说 明
DataSource	提供读取数据库的数据源信息
SimpleJdbcTemplate	提供标准的 Spring 的 jdbc 模板，根据分页信息查询数据库，返回排序后的结果集 ResultSet
PagingQueryProvider	根据分页信息生成每次需要查询的 SQL 语句
RowMapper	负责将结果集 ResultSet 转换为 Java 对象

表 6-18 JdbcPagingItemReader 关键属性

JdbcPagingItemReader 属性	类 型	说 明
dataSource	DataSource	数据源，通过该属性指定使用的数据库信息
fetchSize	int	设置 ResultSet 每次向数据库取的行数；setFetchSize 的意思是当调用 rs.next 时，ResultSet 会一次性从服务器上取多少行数据回来，这样在下次 rs.next 时，可以直接从内存中取出数据而不需要网络交互，提高了效率。 默认值：-1
queryProvider	PagingQueryProvider	分页查询 SQL 语句生成器，负责根据分页信息生成每次需要执行的 SQL 语句
parameterValues	Map<String, Object>	设置定义的 SQL 语句中的参数
rowMapper	RowMapper	将结果集 ResultSet 转换为指定的 Pojo 对象类；需要实现 RowMapper 接口默认，可以使用 BeanPropertyRowMapper；
pageSize	int	分页大小。 默认值：10

Spring Batch 框架为了支持 PagingQueryProvider，根据不同的数据库类型提供多种实现，为了便于开发者屏蔽不同的数据库类型，Spring Batch 框架提供了友好的工厂类 SqlPagingQueryProviderFactoryBean 为不同的数据库类型提供 PagingQueryProvider 的实现类。下面列出工厂类需要的关键属性，参见表 6-19。

表 6-19 SqlPagingQueryProviderFactoryBean 关键属性

SqlPagingQueryProviderFactoryBean 属性	类 型	说 明
dataSource	DataSource	数据源，通过该属性指定使用的数据库信息
databaseType	String	指定数据库的类型，如果不显示指定该类型，则自动通过 dataSource 属性获取数据库的信息
ascending	Boolean	查询语句是否是升序。 默认值：true
fromClause	String	定义查询语句的 from 部分
selectClause	String	定义查询语句的 select 部分
sortKey	String	定义查询语句排序的关键字段
whereClause	String	定义查询语句的 where 字段

配置 JdbcPagingItemReader

使用 JdbcPagingItemReader 至少需要配置 dataSource、queryProvider、rowMapper 三个属性；dataSource 指定访问的数据源，queryProvider 用于定义分页查询的 SQL 语句，rowMapper 用于将结果集 ResultSet 转换为 Java 业务对象。

在给出详细配置文件之前，我们首先准备 SQL（使用的为 MySQL 数据库）脚本，示例中使用信用卡账单表，存放信用卡消费记录情况，主要字段包括 ID、ACCOUNTID、NAME、AMOUNT、DATE、ADDRESS；建表脚本参见代码清单 6-67。

完整内容参见文件 ch06/db/create-tables-mysql.sql。

代码清单 6-67 建表脚本

```
1. CREATE TABLE t_credit
2.     (ID VARCHAR(10),
3.      ACCOUNTID VARCHAR(20),
4.      NAME VARCHAR(10),
5.      AMOUNT NUMERIC(10,2),
6.      DATE VARCHAR(20),
7.      ADDRESS VARCHAR(128),
8.      primary key (ID)
9.  )
10. ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

接下来我们使用分页度配置批处理任务，参见代码清单 6-68。

完整配置文件参见：ch06/job/job-db-paging-jdbc.xml。

代码清单 6-68 配置 JdbcPagingItemReader

```
1. <bean:bean id="jdbcItemPageReader" scope="step"
2.         class="org.springframework.batch.item.database.
           JdbcPagingItemReader">
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="queryProvider" ref="refQueryProvider" />
5.     <bean:property name="parameterValues">
6.         <bean:map>
7.             <bean:entry key="begin" value="#{jobParameters['id_begin']}" />
8.             <bean:entry key="end" value="#{jobParameters['id_end']}" />
9.         </bean:map>
10.    </bean:property>
11.    <bean:property name="pageSize" value="2"/>
12.    <bean:property name="rowMapper" ref="custCreditRowMapper"/>
13. </bean:bean>
14. <bean:bean id="refQueryProvider" class="org.springframework.batch.item.
15. database.support.SqlPagingQueryProviderFactoryBean">
16.     <bean:property name="dataSource" ref="dataSource"/>
17.     <bean:property name="selectClause"
18.         value="select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS"/>
19.     <bean:property name="fromClause" value="from t_credit"/>
20.     <bean:property name="whereClause" value="where ID between :begin
           and :end"/>
21.     <bean:property name="sortKey" value="ID"/>
22. </bean:bean>
```

其中，3 行：属性 `dataSource` 定义需要访问的数据源。

4 行：属性 `queryProvider` 定义需要分页查询的语句，对象“`refQueryProvider`”的具体配置参见 14~22 行，本例中根据 ID 字段所在的范围区间查询表 `t_credit` 中的所有字段信息。

5~10 行：属性 `parameterValues` 定义需要执行 SQL 中的参数

```
<bean:entry key="begin" value="#{jobParameters['id_begin']}" />
```

key 中的值“begin”对应 `queryProvider` 中的变量“`:begin`”；value 中的“`{jobParameters['id_begin']}`”表示使用作业参数“id_begin”作为 begin 的值。

11 行：属性 `pageSize` 用于指定分页的大小，本例指定为 2；如果不指定使用默认值 10。

12 行：属性 `rowMapper` 将结果集 `ResultSet` 转换为 `CreditBill` 对象，使用自定义的 `Mapper` 实现 `custCreditRowMapper`。

配置数据源

配置数据源参见代码清单 6-69。

代码清单 6-69 配置数据源

```
1. <context:property-placeholder location="classpath:/ch06/properties/
   batch-mysql.properties" />
```

```

2. <bean:bean id="dataSource"
3.     class="org.springframework.jdbc.datasource.
        DriverManagerDataSource">
4.     <bean:property name="driverClassName">
5.         <bean:value>${datasource.driver}</bean:value>
6.     </bean:property>
7.     <bean:property name="url">
8.         <bean:value>${datasource.url}</bean:value>
9.     </bean:property>
10.    <bean:property name="username" value="${datasource.username}">
11.        </bean:property>
12.    <bean:property name="password" value="${datasource.password}">
13.        </bean:property>
14. </bean:bean>

```

其中，1 行：指定加载的属性配置文件，方便动态指定数据源的属性。

2~12 行：配置数据源的具体实现，分别指定属性 `driverClassName`、`url`、`username`、`password`。

自定义 RowMapper

使用 `RowMapper` 进行数据映射时，可以使用 Spring 框架提供的 `BeanPropertyRowMapper` 来自动将数据库的字段映射到指定的 Java Bean 上；另外只需要实现接口 `org.springframework.jdbc.core.RowMapper<T>`，根据需要完成自定义的数据转换功能，可以自定义 `RowMapper` 的实现类。`RowMapper` 接口定义参见代码清单 6-70。

代码清单 6-70 `RowMapper` 接口定义

```

1. public interface RowMapper<T> {
2.     T mapRow(ResultSet rs, int rowNum) throws SQLException;
3. }

```

其中，2 行：核心转换方法，参数为结果集和当前游标所在的位置。

本例中使用自定义的信用卡账单转换类 `com.juxtapose.example.ch06.db.CreditBillRowMapper`。`CreditBillRowMapper` 的代码实现参见代码清单 6-71。

代码清单 6-71 `CreditBillRowMapper` 类定义

```

1. public class CreditBillRowMapper implements RowMapper<CreditBill> {
2.
3.     public CreditBill mapRow(ResultSet rs, int rowNum) throws SQLException {
4.         CreditBill bill = new CreditBill();
5.         bill.setAccountID(rs.getString("ACCOUNTID"));
6.         bill.setAddress(rs.getString("ADDRESS"));
7.         bill.setAmount(rs.getDouble("AMOUNT"));
8.         bill.setDate(rs.getString("DATE"));
9.         bill.setName(rs.getString("NAME"));

```

```
10.         return bill;
11.     }
12. }
```

其中，3~11 行：业务实现代码，将给定的结果集 `ResultSet` 转化为 `CreditBill` 对象。
使用代码清单 6-72 执行定义的 `dbPagingReadJob`。
完整代码参见：`test.com.juxtapose.example.ch06.JobLaunchJDBCPaging`。

代码清单 6-72 执行 `dbPagingReadJob`

```
1. executeJob("ch06/job/job-db-paging-jdbc.xml", "dbPagingReadJob",
2.           new JobParametersBuilder().addDate("date", new Date())
3.           .addString("id_begin", "1").addString("id_end", "4"));
```

其中，2 行：传入参数 `id_begin=1`，`id_end=4`，最终 SQL 执行查询的脚本为“`select ID,ACCOUNTID,NAME,AMOUNT,DATE, ADDRESS from t_credit where id between 1 and 4`”。

6.5.5 HibernatePagingItemReader

对象关系映射（Object Relational Mapping，ORM）是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，使得 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。Spring Batch 框架对 ORM 类型的 Hibernate 提供了基于分页的读 `ItemReader`。

HibernatePagingItemReader 实现 `ItemReader` 接口，核心作用是将数据库中的记录通过 ORM 的分页方式转换为 Java Bean 对象。学会使用基于 `HibernateCursorItemReader` 的读数据库后，使用 `HibernatePagingItemReader` 非常简单。

HibernatePagingItemReader 结构及关键属性

HibernatePagingItemReader 的关键支持的属性，参见表 6-20。

表 6-20 HibernatePagingItemReader 关键属性

HibernatePagingItemReader 属性	类 型	说 明
fetchSize	int	设置 <code>ResultSet</code> 每次向数据库取的行数； <code>setFetchSize</code> 的意思是当调用 <code>rs.next</code> 时， <code>ResultSet</code> 会一次性从服务器上取多少行数据 回来，这样在下次 <code>rs.next</code> 时，可以直接从内 存中取出数据而不需要网络交互，提高了效 率。 默认值：-1

续表

HibernatePagingItemReader 属性	类 型	说 明
maxItemCount	int	设置结果集做大行数。 默认值: Integer.MAX_VALUE
parameterValues	String	Statement 的参数值
queryProvider	HibernateQueryProvider	生成 HQL 的查询类
queryString	String	HQL 查询语句
sessionFactory	SessionFactory	Hibernate 的 SessionFactory, 负责与数据库进行交互
useStatelessSession	Boolean	是否使用无状态的会话。 默认值: true
pageSize	int	分页大小。 默认值: 10

配置 HibernatePagingItemReader

使用 HibernatePagingItemReader 至少需要配置 sessionFactory、queryString 两个属性; sessionFactory 是 Hibernate 的会话工厂类, 用于与数据库进行交互访问; queryString 用于指定查询的 HQL 语句。

在给出详细配置文件之前, 我们首先准备实体对象 (参见代码清单 6-73)、配置 Hibernate 的.cfg.xml (参见代码清单 6-74)。本节示例仍然使用 6.5.1 章节使用的数据库脚本。

配置数据实体对象

完整内容参见类: com.juxtapose.example.ch06.hibernate.CreditBill。

代码清单 6-73 数据实体对象 CreditBill

```

1. @Entity
2. @Table(name="t_credit")
3. public class CreditBill {
4.     @Id
5.     @Column(name="ID")
6.     private String id;
7.
8.     @Column(name="ACCOUNTID")
9.     private String accountID = "";    /** 银行卡账户 ID */
10.
11.     @Column(name="NAME")
12.     private String name = "";        /** 持卡人姓名 */
13.
14.     @Column(name="AMOUNT")
15.     private double amount = 0;       /** 消费金额 */

```

```

16.
17.     @Column(name="DATE")
18.     private String date;           /** 消费日期，格式 YYYY-MM-DD HH:MM:SS*/
19.
20.     @Column(name="ADDRESS")
21.     private String address;        /** 消费场所 */
22.
23.     .....
24. }

```

其中，1 行：@Entity 注释声明该类为持久类，将一个 JavaBean 类声明为一个实体的数据库表映射类，建议实现序列化。默认情况下，所有的类属性都为映射到数据表的持久性字段。若在类中，添加另外属性，而非映射来数据库的，要用 Transient 来注解。

2 行：@Table(name="t_credit")持久性映射的表，表名为“t_credit”。@Table 是类一级的注解，定义在@Entity之下，为实体 Bean 映射表，目录和 schema 的名字，默认为实体 Bean 的类名，不包含包名。

4 行：@Id，用于标识数据表的主键。

5 行：@Column(name="ID")表示属性对应的数据库列的字段名。

配置 hibernate 的 cfg 文件

数据实体配置完成后，需要配置 hibernate 的配置文件，用于声明上面的数据实体。

完整文件参见：ch06/cfg/hibernate.cfg.xml。

代码清单 6-74 配置 hibernate.cg.xml

```

1. <hibernate-configuration>
2.     <session-factory>
3.         <mapping class="com.juxtapose.example.ch06.hibernate.CreditBill"/>
4.     </session-factory>
5. </hibernate-configuration>

```

其中，3 行：声明 Hibernate 中使用的数据实体类 CreditBill。

配置 HibernatePagingItemReader

完整配置文件参见：ch06/job/job-db-paging-hibernate.xml。

代码清单 6-75 展示了从数据库表 t_credit 中读取数据。

代码清单 6-75 配置 HibernatePagingItemReader

```

1.     <bean:bean id="hibernatePagingItemReader" scope="step"
2.         class="org.springframework.batch.item.database.
           HibernatePagingItemReader">
3.         <bean:property name="sessionFactory" ref="sessionFactory"/>
4.         <bean:property name="queryString" value="from CreditBill
5.                               where id between :begin and :end"/>

```

```

6.         <bean:property name="parameterValues">
7.             <bean:map>
8.                 <bean:entry key="begin" value="#{jobParameters['id_begin']}" />
9.                 <bean:entry key="end" value="#{jobParameters['id_end']}" />
10.            </bean:map>
11.        </bean:property>
12.        <bean:property name="pageSize" value="2"/>
13.    </bean:bean>
14.
15.    <bean:bean id="sessionFactory"
16.        class="org.springframework.orm.hibernate3.
17.        LocalSessionFactoryBean">
18.        <bean:property name="dataSource" ref="dataSource"/>
19.        <bean:property name="configurationClass"
20.            value="org.hibernate.cfg.AnnotationConfiguration"/>
21.        <bean:property name="configLocation"
22.            value="classpath:/ch06/cfg/hibernate.cfg.xml"/>
23.        <bean:property name="hibernateProperties">
24.            <bean:value>
25.                hibernate.dialect=org.hibernate.dialect.MySQLDialect
26.                hibernate.show_sql=true
27.            </bean:value>
28.        </bean:property>
29.    </bean:bean>

```

其中，3 行：属性 sessionFactory，定义 Hibernate 的会话访问工厂类，用于 ORM 映射，访问数据库表信息。

4~5 行：属性 queryString 定义需要查询的 HQL 语句，“from CreditBill where id between :begin and :end”，该 HQL 语句有两个查询参数，分别是:begin 和:end；需要通过属性 parameterValues 指定参数值。

6~11 行：属性 parameterValues 用于指定属性 queryString 中的变量:begin 和:end；此处使用了参数后绑定技术，在 Job 执行时候可以通过指定 JobParameter 给 begin 和 end 参数赋值。

12 行：属性 pageSize 定义分页的大小。

15~28 行：声明 hibernate 使用的会话工厂，使用 hibernate 提供的 LocalSessionFactoryBean，需要为下面的属性赋值 dataSource（数据源），configurationClass（通过注解的方式获取），configLocation（配置文件地址），hibernateProperties（基本属性信息）。

使用代码清单 6-76 中的代码执行定义的 hibernatePagingReadJob。

完整代码参见：test.com.juxtapose.example.ch06.JobLaunchHibernatePaging。

代码清单 6-76 执行 hibernatePagingReadJob

```

1. executeJob("ch06/job/job-db-paging-hibernate.xml",
2.     "hibernatePagingReadJob",

```

```

2.         new JobParametersBuilder().addDate("date", new Date())
3.         .addString("id_begin", "1").addString("id_end", "4"));

```

其中，3 行：传入参数 id_begin=1, id_end=4，最终 SQL 执行查询的脚本为 “select ID, ACCOUNTID,NAME,AMOUNT,DATE, ADDRESS from t_credit where id between 1 and 4”。

6.5.6 JpaPagingItemReader

对象关系映射（Object Relational Mapping，ORM）是一种为解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

JPA（Java Persistence API）是 Sun 官方提出的 Java 持久化规范。JPA 通过 JDK 5.0 注解或 XML 描述对象—关系表的映射关系，并将运行期的实体对象持久化到数据库中；它为 Java 开发人员提供了一种对象/关系映射工具来管理 Java 应用中的关系数据。Spring Batch 框架对 ORM 类型的 JPA 提供了基于分页的读 ItemReader。

JpaPagingItemReader 实现 ItemReader 接口，核心作用将数据库中的记录通过 ORM 的分页方式转换为 Java Bean 对象。学会使用基于 HibernatePagingItemReader 的读数据库后，使用 JpaPagingItemReader 非常简单。下面我们先看 JpaPagingItemReader 的关键支持的属性，参见表 6-21。

JpaPagingItemReader 结构及关键属性

表 6-21 JpaPagingItemReader 关键属性

JpaPagingItemReader 属性	类 型	说 明
maxItemCount	int	设置结果集做大行数。 默认值：Integer.MAX_VALUE
parameterValues	String	执行 SQL 的参数值
queryProvider	JpaQueryProvider	生成 JPQL 的查询类
queryString	String	JPQL 查询语句
entityManagerFactory	EntityManagerFactory	用于创建实体管理器
pageSize	int	分页大小 默认值：10

配置 JpaPagingItemReader

使用 JpaPagingItemReader 至少需要配置 entityManagerFactory、queryString 两个属性；entityManagerFactory 负责创建 EntityManager，后者负责完成对实体的增删改查等操作；queryString 用于指定查询的 JPQL 语句。

在给出详细配置文件之前，我们首先准备实体对象（参见代码清单 6-77）、配置 JPA 的

实体映射文件（参见代码清单 6-78）。本节示例仍然使用 6.5.1 章节使用的数据库脚本。

配置数据实体对象

完整内容参见类：`com.juxtapose.example.ch06.jpa.CreditBill`。

代码清单 6-77 数据实体对象 `CreditBill`

```
1. @Entity
2. @Table(name="t_credit")
3. public class CreditBill {
4.     @Id
5.     @Column(name="ID")
6.     private String id;
7.
8.     @Column(name="ACCOUNTID")
9.     private String accountID = "";    /** 银行卡账户 ID */
10.
11.    @Column(name="NAME")
12.    private String name = "";        /** 持卡人姓名 */
13.
14.    @Column(name="AMOUNT")
15.    private double amount = 0;       /** 消费金额 */
16.
17.    @Column(name="DATE")
18.    private String date;              /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */
19.
20.    @Column(name="ADDRESS")
21.    private String address;          /** 消费场所 */
22.
23.    .....
24. }
```

其中，1 行：`@Entity` 注释声明该类为持久类，将一个 `JavaBean` 类声明为一个实体的数据库表映射类，建议实现序列化。默认情况下，所有的类属性都为映射到数据表的持久性字段。若在类中，添加另外属性，而非映射自数据库的，要用 `Transient` 来注解。

2 行：`@Table(name="t_credit")` 持久性映射的表，表名为“`t_credit`”。`@Table` 是类一级的注解，定义在 `@Entity` 之下，为实体 `Bean` 映射表、目录和 `schema` 的名字，默认为实体 `Bean` 的类名，不包含包名。

4 行：`@Id`，用于标识数据表的主键。

5 行：`@Column(name="ID")` 表示属性对应的数据库列的字段名。

配置 JPA 的持久化文件

数据实体配置完成后，需要配置 JPA 的实体持久化配置文件，用于声明上面的数据实体。

完整文件参见：ch06/jpa/persistence.xml。

代码清单 6-78 JPA 持久化配置 persistence.xml

```
1. <persistence>
2.     <persistence-unit name="creditBill" transaction-type="RESOURCE_LOCAL">
3.         <class>com.juxtapose.example.ch06.jpa.CreditBill</class>
4.         <exclude-unlisted-classes>true</exclude-unlisted-classes>
5.     </persistence-unit>
6. </persistence>
```

其中，3 行：声明 JPA 中使用的数据实体类 CreditBill。

4 行：声明排除所有未在此声明的实体类。

配置 JpaPagingItemReader

完整配置文件参见：ch06/job/job-db-paging-jpa.xml。

代码清单 6-79 展示了从数据库表 t_credit 中读取数据。

代码清单 6-79 配置 JpaPagingItemReader

```
1. <bean:bean id="jpaPagingItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
3.         JpaPagingItemReader">
4.     <bean:property name="entityManagerFactory"
5.         ref="entityManagerFactory" />
6.     <bean:property name="queryString"
7.         value="select c from CreditBill c where id between :begin
8.             and :end" />
9.     <bean:property name="parameterValues">
10.         <bean:map>
11.             <bean:entry key="begin" value="#{jobParameters['id_begin']}" />
12.             <bean:entry key="end" value="#{jobParameters['id_end']}" />
13.         </bean:map>
14.     </bean:property>
15.     <bean:property name="pageSize" value="2"/>
16. </bean:bean>
```

其中，3 行：属性 entityManagerFactory，定义 JPA 的实体管理器对象（参见代码清单 6-80），提供访问数据库表的操作。

4~5 行：属性 queryString 定义需要查询的 JPQL 语句，“select c from CreditBill c where id between :begin and :end”，该 JPQL 语句有两个查询参数，分别是:begin 和:end；需要通过属性 parameterValues 指定参数值。

6~11 行：属性 parameterValues 用于指定属性 queryString 中的变量:begin 和:end；此处使用了参数后绑定技术，在 Job 执行时候可以通过指定 JobParameter 给 begin 和 end 参数赋值。

12 行：属性 pageSize 定义分页的大小。

配置实体管理器对象

代码清单 6-80 配置实体管理器对象

```
1. <bean:bean id="entityManagerFactory"
2.     class="org.springframework.orm.jpa.
        LocalContainerEntityManagerFactoryBean">
3.     <bean:property name="dataSource" ref="dataSource" />
4.     <bean:property name="persistenceUnitName" value="creditBill" />
5.     <bean:property name="persistenceXmlLocation"
6.         value="classpath:/ch06/jpa/persistence.xml" />
7.     <bean:property name="jpaVendorAdapter">
8.         bean:bean class="org.springframework.orm.jpa.vendor.
9.             HibernateJpaVendorAdapter">
10.         <bean:property name="showSql" value="true" />
11.     </bean:bean>
12. </bean:property>
13. <bean:property name="jpaDialect">
14.     <bean:bean class="org.springframework.orm.jpa.vendor.
        HibernateJpaDialect" />
15. </bean:property>
16. </bean:bean>
```

其中，5 行：属性 `persistenceXmlLocation` 指定需要加载的持久化配置文件。

7~11 行：属性 `jpaVendorAdapter` 指定具体的 Provider，此处使用 Hibernate 的实现。

使用代码清单 6-81 执行定义的 `jpaPagingReadJob`。

完整代码参见：`test.com.jxtapose.example.ch06.JobLaunchJpaPaging`。

代码清单 6-81 执行 `jpaPagingReadJob`

```
1. executeJob("ch06/job/job-db-paging-jpa.xml", "jpaPagingReadJob",
2.     new JobParametersBuilder().addDate("date", new Date())
3.     .addString("id_begin", "1").addString("id_end", "4"));
```

其中，3 行：传入参数 `id_begin=1`，`id_end=4`，最终 SQL 执行查询的脚本为“`select ID, ACCOUNTID, NAME, AMOUNT, DATE, ADDRESS from t_credit where id between 1 and 4`”。

6.5.7 IbatisPagingItemReader

对象关系映射（Object Relational Mapping，ORM）是一种为解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

Ibatis 是一个开放源代码的对象关系映射框架，相对于 Hibernate，Itabis 对 JDBC 进行了更轻量级的对象封装，由于 Iabtis 支持编写 SQL，使得 Ibatis 框架具有更高的灵活性和性能。Spring Batch 框架对 Ibatis 提供了基于分页的读 `ItemReader`。

IbatisPagingItemReader 实现 ItemReader 接口，其核心作用将数据库中的记录通过 ORM 的分页方式转换为 Java Bean 对象。学会使用基于 HibernatePagingItemReader 的读数据库后，使用 IbatisPagingItemReader 非常简单。下面我们先看 IbatisPagingItemReader 的关键支持的属性，参见表 6-22。

IbatisPagingItemReader 结构及关键属性

表 6-22 IbatisPagingItemReader 关键属性

IbatisPagingItemReader 属性	类 型	说 明
sqlMapClient	SqlMapClient	用于指定执行的命名 SQL 的配置文件和数据源
queryId	String	命名 SQL 定义的 ID
sqlMapClientTemplate	SqlMapClientTemplate	命名 SQL 执行的默认模板
parameterValues	Map<String, Object>	设置定义的 SQL 语句中的参数
pageSize	int	分页大小 默认值：10

配置 IbatisPagingItemReader

使用 IbatisPagingItemReader 至少需要配置 sqlMapClient、queryId 两个属性；sqlMapClient 用于指定配置的命名 SQL 的文件；queryId 用于指定命名 SQL 文件中定义的 SQL 语句。

在给出详细配置文件之前，我们首先准备配置命名 SQL 的配置文件和命名 SQL 文件。本节示例仍然使用 6.5.1 章节使用的数据库脚本。

定义命名 SQL 文件

命名 SQL 框架提供了标准的格式用于定义命名 SQL 文件。代码清单 6-82 的命名 SQL 对表 t_credit 进行操作：提供 ID 为“getAllCredits”的命名 SQL 用于查询所有的信用卡账单对象；提供 ID 为“getCreditsById”的命名 SQL 用于根据 ID 所在的区间查询信用卡账单对象。

完整文件参见：ch06/ibatis/ibatis-credit.xml。

代码清单 6-82 命名 SQL 定义

```
1. <sqlMap namespace="Credit">
2.   <resultMap id="result" class="com.juxtapose.example.ch06.ibatis.CreditBill">
3.     <result property="id" column="ID" />
4.     <result property="accountID" column="ACCOUNTID" />
5.     <result property="name" column="NAME" />
6.     <result property="amount" column="AMOUNT" />
7.     <result property="date" column="DATE" />
8.     <result property="address" column="ADDRESS" />
9.   </resultMap>
10.
11. <!-- 查询所有的信用卡账单对象 -->
```

```

12. <select id="getAllCredits" resultMap="result">
13.     select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS from t_credit
14. </select>
15.
16. <!-- 根据 ID 所在区间查询信用卡账单对象 -->
17. <select id="getCreditsById" parameterClass="java.util.HashMap"
18.     resultMap="result">
19.     select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS
20.         from t_credit where ID between #begin# and #end#
21. </select>
22. </sqlMap>

```

其中，1 行：声明当前命名 SQL 文件的命名空间为“Credit”。

2~9 行：定义命名 SQL 的返回值对象，将数据库的列于对象 `com.juxtapose.example.ch06.ibatis.CreditBill` 的属性进行映射，命名 SQL 执行后会自动将一行转换为指定的对象；以 `<result property="id" column="ID" />` 举例：命名 SQL 执行后，会将表 `t_credit` 中的字段 `ID` 映射到对象 `com.juxtapose.example.ch06.ibatis.CreditBill` 的 `ID` 属性上。

12~14 行：提供 ID 为“`getAllCredits`”的命名 SQL 用于查询所有的信用卡账单对象。

16~20 行：提供 ID 为“`getCreditsById`”的命名 SQL 用于根据 ID 所在的区间查询信用卡账单对象，此处定义了参数 `#begin#` 和 `#end#`，通过属性 `parameterClass` 定义参数的类型为“`java.util.HashMap`”。

配置命名 SQL 的配置文件

命名 SQL 文件定义完成后，需要在 Ibatis 的配置文件中声明参见代码清单 6-83；然后可以在 `IbatisPagingItemReader` 中使用命名 SQL 执行数据库的查询，参见代码清单 6-84。

完整文件参见：ch06/ibatis/ibatis-config.xml。

代码清单 6-83 声明命名 SQL 配置

```

1. <sqlMapConfig>
2.     <sqlMap resource="ch06/ibatis/ibatis-credit.xml"/>
3. </sqlMapConfig>

```

其中，2 行：声明具体的 Ibatis 的命名 SQL 文件。

配置 IbatisPagingItemReader

完整配置文件参见：ch06/job/job-db-paging-ibatis.xml。

代码清单 6-84 展示了通过命名 SQL 的方式从数据库表 `t_credit` 中读取数据。

代码清单 6-84 配置 IbatisPagingItemReader

```

1. <bean:bean id="ibatisPagingItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
3.         IbatisPagingItemReader">

```

```

3.      <!-- <bean:property name="queryId" value="getAllCredits" /> -->
4.      <bean:property name="queryId" value="getCreditsById" />
5.      <bean:property name="sqlMapClient" ref="sqlMapClient" />
6.      <bean:property name="parameterValues">
7.          <bean:map>
8.              <bean:entry key="begin"
9.                  value="#{jobParameters['id_begin']}" />
10.             <bean:entry key="end" value="#{jobParameters['id_end']}" />
11.          </bean:map>
12.      </bean:property>
13.  </bean:bean>
14.  <bean:bean id="sqlMapClient"
15.      class="org.springframework.orm.ibatis.
16.          SqlMapClientFactoryBean">
17.      <bean:property name="dataSource" ref="dataSource" />
18.      <bean:property name="configLocation"
19.          value="classpath:/ch06/ibatis/ibatis-config.xml" />
20.  </bean:bean>

```

其中，4 行：属性 queryId，定义访问的命名 SQL 的 ID，此处访问 ID 为 getCreditsById 的命名 SQL 语句。

5 行：属性 sqlMapClient 定义命名 SQL 的客户端配置文件。

6～11 行：属性 parameterValues 用于指定属性命名 SQL 中的变量 #begin# 和 #end#；此处使用了参数后绑定技术，在 Job 执行时候可以通过指定 JobParameter 给 id_begin 和 id_end 参数赋值。

14～19 行：给出了 sqlMapClient 对象的具体定义，需要为其指定 2 个属性分别是 dataSource 和 configLocation；dataSource 指定使用的数据源，configLocation 指定命名 SQL 的配置文件加载地址。

使用代码清单 6-85 执行定义的.ibatisPagingReadJob。

完整代码参见：test.com.juxtapose.example.ch06.JobLaunchIbatisPaging。

代码清单 6-85 执行.ibatisPagingReadJob

```

1. executeJob("ch06/job/job-db-paging-ibatis.xml", "ibatisPagingReadJob",
2.     new JobParametersBuilder().addDate("date", new Date())
3.     .addString("id_begin", "1").addString("id_end", "4"));

```

其中，3 行：传入参数 id_begin=1，id_end=4，最终 SQL 执行查询的脚本为“select ID,ACCOUNTID,NAME,AMOUNT,DATE, ADDRESS from t_credit where id between 1 and 4”。

6.6 读 JMS 队列

JMS (Java Messaging Service) 是 Java 平台上有关面向消息中间件 (MOM) 的技术规范, 它便于消息系统中的 Java 应用程序进行消息交换, 并且通过提供标准的产生、发送、接收消息的接口。JMS 是一种与厂商无关的 API, 用来访问消息收发系统消息。它类似于 JDBC (Java Database Connectivity), JDBC 是可以用来访问许多不同关系数据库的 API, 而 JMS 则提供同样与厂商无关的访问方法, 以访问消息收发服务。

Spring 的 JMS 抽象框架简化了 JMS API 的使用, 并与 JMS 提供者 (比如 IBM 的 WebSphere MQ、开源的 ActiveMQ 等) 平滑地集成。Spring JMS 框架提供了 JMS 访问的模板类 `JmsTemplate`, 模板类处理资源的创建和释放, 简化了 JMS 的使用。Spring Batch 框架基于 Spring JMS 框架提供了对 JMS 队列读取的 `ItemReader`。

6.6.1 JmsItemReader

`JmsItemReader` 实现 `ItemReader` 接口, 核心作用将 JMS 队列中的消息转换为 Java 对象。`JmsItemReader` 引用 `JmsOperations`, 后者负责对 JMS 队列消息的进行读取, 并转化为指定的类型。

JmsItemReader 结构关键属性

图 6-21 展示了 JMS 队列读取的逻辑架构图, `JmsOperations` 负责将消息从队列中读取, 并按照指定的消息类型格式转换为 Java 对象。

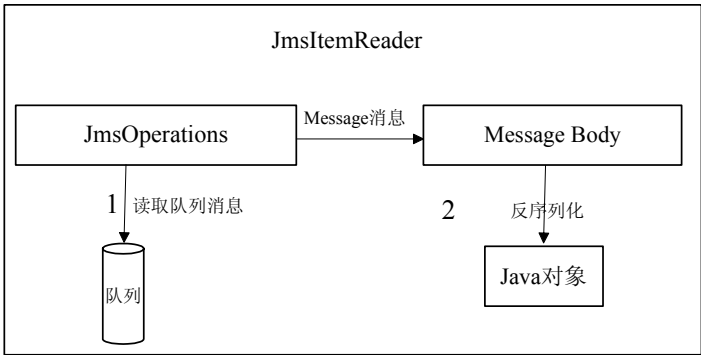


图 6-21 JMS 队列读取的逻辑架构

`JmsItemReader` 核心类架构图参见图 6-22。

`JmsItemReader` 将队列读取全部代理给 `JmsOperations`, `JmsOperations` 读取队列消息后根据指定的消息类型将消息的 Body 部分转换为属性 `itemType` 指定的对象。

`JmsItemReader` 关键属性参见表 6-23。

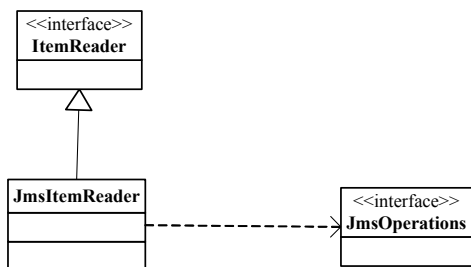


图 6-22 JmsItemReader 核心类

表 6-23 JmsItemReader 关键属性

JmsItemReader 属性	类 型	说 明
itemType	Class	Item 对象的类型
jmsTemplate	JmsOperations	消息发送模板

配置 JmsItemReader

现在假设所有的信用卡账单都是通过 JMS 消息的方式从外部系统发送过来的，需要使用 Spring Batch 框架从 JMS 队列中读取信用卡消费列表。配置 JmsItemReader 非常简单，只需要指定两个属性 itemType 和 jmsTemplate 即可，参见代码清单 6-86。本示例使用 Apache 的 ActiveMQ 作为消息中间件。

完整配置参见文件 ch06/job/job-jms.xml。

代码清单 6-86 配置 JmsItemReader

```

1.  <!-- 读取 jms -->
2.      <bean:bean id="jmsItemReader"
3.          class="org.springframework.batch.item.jms.JmsItemReader">
4.          <bean:property name="itemType" value="com.juxtapose.example.ch06.
            CreditBill"/>
5.          <bean:property name="jmsTemplate" ref="jmsTemplate"/>
6.      </bean:bean>
  
```

其中，4 行：属性 itemType 执行消息队列中存放的对象类型。

5 行：属性 jmsTemplate 指定读取 JMS 消息的模板，用于读取指定队列中的消息；消息模板配置参见代码清单 6-87。

JMS 消息模板的配置

代码清单 6-87 配置 JMS 消息模板

```

1.      <bean:bean id="jmsTemplate" class="org.springframework.jms.core.
            JmsTemplate">
2.          <bean:property name="connectionFactory" ref="jmsFactory"/>
3.          <bean:property name="defaultDestination" ref="creditDestination"/>
  
```



```

4.      <bean:property name="receiveTimeout" value="500"/>
5.    </bean:bean>
6.
7.    <amq:broker useJmx="false" persistent="false" schedulerSupport="false">
8.      <amq:transportConnectors>
9.        <amq:transportConnector uri="tcp://localhost:61616" />
10.      </amq:transportConnectors>
11.    </amq:broker>
12.
13.    <amq:connectionFactory id="jmsFactory" brokerURL="tcp://localhost:
14.      61616"/>
15.    <amq:queue id="creditDestination" physicalName="destination.
16.      creditBill" />

```

其中，2 行：属性 `connectionFactory` 用于配置 JMS 的连接工厂。

3 行：属性 `defaultDestination` 指定需要读取的目标消息队列。

4 行：属性 `receiveTimeout` 指定读取消息的超时时间。

7~11 行：定义 AMQ 的 broker，提供 JMS 的服务器端，指定对应的 ip 与 port；属性 `persistent` 表示不让消息持久化；属性 `schedulerSupport` 设置为 `false`，禁止掉 AMQ 的延迟发送的功能，可避免因为 AMQ 异常终止后导致无法启动。

13 行：定义 JMS 的连接工厂。

14 行：定义消息存储的队列，AMQ 启动时会自动创建名字为 `"destination.creditBill"` 的队列。

由于引用了新的命名空间 `amq`，需要在头文件中定义命名空间，参见代码清单 6-88。

代码清单 6-88 引入 `amq` 命名空间

```

1. <bean:beans xmlns="
2.   xmlns:amq="http://activemq.apache.org/schema/core"
3.   xsi:schemaLocation="
4.     http://activemq.apache.org/schema/core
5.     http://activemq.apache.org/schema/core/activemq-core.xsd">
6.   .....
7. </bean:beans>

```

作业 Job 配置完成后，为了能从指定的消息队列读取数据，需要将消息首先发送到队列中；使用 `JMSTemplate` 可以方便地读、写消息。代码清单 6-89 给出将消息发送到队列的代码片段。

完整代码参见：test.com/juxtapose.example.ch06.JobLaunchJMS。

代码清单 6-89 将消息发送到队列

```

1. public static ApplicationContext getContext(String jobPath){
2.   return new ClassPathXmlApplicationContext(jobPath);
3. }

```

```

4.
5.  public static JmsTemplate getJmsTemplate(ApplicationContext context){
6.      JmsTemplate jmsTemplate = (JmsTemplate)context.getBean
7.          ("jmsTemplate");
8.      return jmsTemplate;
9.  }
10. public static void sendMessage(JmsTemplate jmsTemplate, final CreditBill
    creditBill){
11.     jmsTemplate.send(new MessageCreator() {
12.         public Message createMessage(Session session)
13.             throws JMSEException {
14.             ObjectMessage message = session.createObjectMessage();
15.             message.setObject(creditBill);
16.             return message;
17.         }
18.     });
19. }
20.
21. //示例发送消息代码
22. //将消息发送到定义的队列"destination.creditBill"中
23. ApplicationContext context = getContext("ch06/job/job-jms.xml");
24. JmsTemplate jmsTemplate = getJmsTemplate(context);
25. sendMessage(jmsTemplate,
26.     new CreditBill("4047390012345678","tom",100.00,"2013-2-2 12:00:08",
27.         "Lu Jia Zui road"));
28. sendMessage(jmsTemplate,
29.     new CreditBill("4047390012345678","tom",320,"2013-2-3 10:35:21","Lu
30.         Jia Zui road"));
31. sendMessage(jmsTemplate,
32.     new CreditBill("4047390012345678","tom",360.00,"2013-2-11 11:12:38",
33.         "Longyang road"));

```

其中，1~3 行：定义获取 Spring Context 上下文方法。

5~8 行：定义从 Spring Context 获取 JMS 模板的方法。

10~19 行：使用给定的 JmsTemplate 发送给定的 CreditBill 消息到队列中；发送的消息类型为 Object 类型消息。

22~30 行：使用上面提供的方法发送 3 条消息到队列"destination.creditBill"中。

使用代码清单 6-90 执行定义的 jmsReadJob。

完整代码参见：test.com.juxtapose.example.ch06.JobLaunchJMS。

代码清单 6-90 执行 jmsReadJob

```

1.  ApplicationContext context = getContext("ch06/job/job-jms.xml");
2.  JmsTemplate jmsTemplate = getJmsTemplate(context);

```

```

3.  sendMessage(jmsTemplate,
4.  new CreditBill("4047390012345678","tom",100.00,"2013-2-2 12:00:08",
   "Lu Jia Zui road"));
5.  sendMessage(jmsTemplate,
6.  new CreditBill("4047390012345678","tom",320,"2013-2-3 10:35:21","Lu
   Jia Zui road"));
7.  sendMessage(jmsTemplate,
8.  new CreditBill("4047390012345678","tom",360.00,"2013-2-11 11:12:38",
   "Longyang road"));
9.  executeJob(context, "jmsReadJob",
10.      new JobParametersBuilder().addDate("date", new Date()));

```

其中，9~10 行：执行"jmsReadJob"的作业，将消息从队列"destination.creditBill"中读出。

6.7 服务复用

复用现有的企业资产和服务是提高企业应用开发的快捷手段，Spring Batch 框架的读组件提供了复用现有服务的能力，利用 Spring Batch 框架提供的 `ItemReaderAdapter` 可以方便地复用业务服务、Spring Bean、EJB 或者其他远程服务。

ItemReaderAdapter 结构关键属性

`ItemReaderAdapter` 持有服务对象，并调用指定的操作来完成 `ItemReader` 中定义的 `read` 功能。需要注意的是：`ItemReader` 的 `read` 操作需要每次返回一条对象，当没有数据可以读取时需要返回 `null`；而现有的服务通常返回一个对象的数组或者 `List` 列表；因此现有的服务通常不能直接被 `ItemReaderAdapter` 使用，这需要在 `ItemReaderAdapter` 和现存的服务之间再增加一个 `ServiceAdapter`（服务适配器）来完成适配工作。

`ItemReaderAdapter`、服务适配（`ServiceAdapter`）和现有服务之间的关系参见图 6-23。

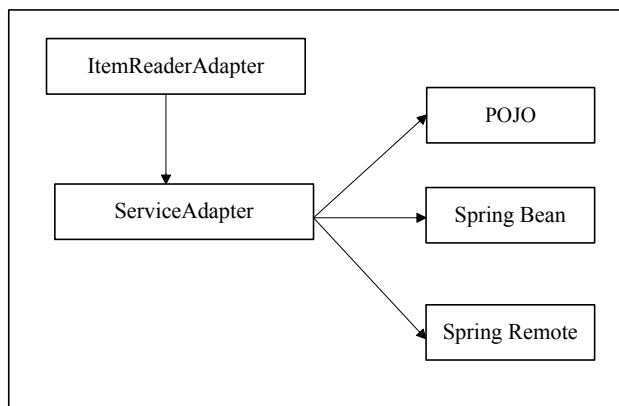


图 6-23 `ItemReaderAdapter`、服务适配（`ServiceAdapter`）和现有服务之间的关系

ItemReaderAdapter 通过 ServiceAdapter 来完成对现有服务的复用，ServiceAdapter 通常封装现存的服务例如 POJO、Spring Bean、Spring Remote 服务等满足 ItemReader 中定义的 read 操作要求。表 6-24 给出了 ItemReaderAdapter 的关键属性。

ItemReaderAdapter 关键属性

表 6-24 ItemReaderAdapter 关键属性

ItemReaderAdapter 属性	类 型	说 明
targetObject	Object	需要调用的目标服务对象
targetMethod	String	需要调用的目标操作名称
Arguments	Object[]	需要调用的操作参数

在配置 ItemReaderAdapter 时候只需要指定上面的三个属性即可，其中前两个参数 targetObject 和 targetMethod 必须填写，参数 arguments 根据操作是否有参数进行选择。

配置 ItemReaderAdapter

目前已经有服务（com.juxtapose.example.ch06.reuse.ExistService）能够查询所有的信用卡账单信息，接下来我们使用 ExistService 作为示例来演示如何使用 ItemReaderAdapter。

已经存在的服务 ExistService 的示例代码参见代码清单 6-91。

完整代码参见：com.juxtapose.example.ch06.reuse.ExistService。

代码清单 6-91 已存在服务 ExistService

```
1. public class ExistService {
2.     public List<CreditBill> queryAllCreditBill() {
3.         List<CreditBill> list = new ArrayList<CreditBill>();
4.         //business service fill list
5.         return list;
6.     }
7. }
```

操作 queryAllCreditBill 获取所有的账单信息；我们前面提到已有的服务其对应的返回值需要是 ItemReader 中 read 操作返回的类型；而 queryAllCreditBill 操作返回的是 List 类型，两者之间不匹配。因此我们需要在 ItemReaderAdapter 和现有的服务 ExistService 之间增加一个 Adapter 来完成转换。

新增类 CreditBillServiceAdapter，负责完成 ExistService 和 ItemReader 之间的适配功能。

CreditBillServiceAdapter 的示例代码参见代码清单 6-92。

完整代码参见：com.juxtapose.example.ch06.reuse.CreditBillServiceAdapter。

代码清单 6-92 CreditBillServiceAdapter 类定义

```
1. public class CreditBillServiceAdapter implements InitializingBean{
2.     private ExistService existService;
```

```

3.     List<CreditBill> creditBillList;
4.
5.     @Override
6.     public void afterPropertiesSet() throws Exception {
7.         this.creditBillList = existService.queryAllCreditBill();
8.     }
9.
10.    public CreditBill nextCreditBill() {
11.        if (creditBillList.size() > 0) {
12.            return creditBillList.remove(0);
13.        } else {
14.            return null;
15.        }
16.    }
17.
18.    .....
19. }

```

CreditBillServiceAdapter 通过持有 ExistService 对象，并提供新的操作 nextCreditBill 来完成服务 ExistService 和 ItemReader 之间的适配。操作 nextCreditBill 每次返回一条信用卡账单信息，直到查询完毕返回 null 为止。

操作 afterPropertiesSet 在 Spring 组装对象之后，调用 existService 查询所有的账单信息。

接下来我们学习如何配置 ItemReaderAdapter，参见代码清单 6-93。

完整配置参见文件：ch06/job/job-reuse-service.xml。

代码清单 6-93 配置 ItemReaderAdapter

```

1.     <bean:bean id="reuseServiceRead"
2.         class="org.springframework.batch.item.adapter.
3.             ItemReaderAdapter">
4.         <bean:property name="targetObject" ref="existServiceAdapter"/>
5.         <bean:property name="targetMethod" value="nextCreditBill"/>
6.     </bean:bean>
7.
8.     <bean:bean id="existServiceAdapter"
9.         class="com.juxtapose.example.ch06.reuse.
10.            CreditBillServiceAdapter">
11.         <bean:property name="existService" ref="existService" />
12.     </bean:bean>
13.
14.     <bean:bean id="existService"
15.         class="com.juxtapose.example.ch06.reuse.ExistService" >
16.     </bean:bean>

```

其中，1~5 行：复用现有的服务完成 ItemReaderAdapter 的配置，属性 targetObject 使用

定义的服务适配器 `existServiceAdapter`；属性 `targetMethod` 指定调用 `existServiceAdapter` 的操作 `nextCreditBill`。

7~10 行：定义服务适配器 `existServiceAdapter`，该对象完成了 `ItemReaderAdapter` 和 `existService` 之间的适配。

12~14 行：声明现存的服务。

截至目前我们配置完了如何使用现有的服务，通过 `ItemReaderAdapter` 可以轻松方便地使用现有的服务功能，避免重复发明新的轮子。

使用代码清单 6-94 执行定义的 `reuseServiceReadJob`。

完整代码参见：`test.com.juxtapose.example.ch06.JobLaunchReuseServiceRead`。

代码清单 6-94 执行 `reuseServiceReadJob`

```
1. executeJob("ch06/job/job-reuse-service.xml", "reuseServiceReadJob",
2.           new JobParametersBuilder().addDate("date", new Date()));
```

6.8 自定义 `ItemReader`

Spring Batch 框架提供丰富的 `ItemReader` 组件，当这些默认的系统组件不能满足需求时，我们可以自己实现 `ItemReader` 接口，完成需要的业务操作。自定义实现 `ItemRead` 非常容易，只需要实现接口 `ItemReader`；通常只实现接口 `ItemReader` 的读不支持重启，为了支持可重启的自定义 `ItemReader` 需要实现接口 `ItemStream`。接下来我们展示如何自定义 `ItemReader`。

6.8.1 不可重启 `ItemReader`

接口 `ItemReader` 的定义参见代码清单 6-95。

代码清单 6-95 `ItemReader` 接口定义

```
1. public interface ItemReader<T> {
2.     T read() throws Exception,
3.         UnexpectedInputException, ParseException,
4.         NonTransientResourceException;
5. }
```

在 `read` 操作中实现对指定资源的读取，需要注意的是 `read` 每次返回一条记录，直到没有数据记录返回 `null` 为止。代码清单 6-96 展示了自定义 `CustomCreditBillItemReader` 的实现。

完整代码参见：`com.juxtapose.example.ch06.cust.itemreader.CustomCreditBillItemReader`。

代码清单 6-96 `CustomCreditBillItemReader` 类定义

```
1. public class CustomCreditBillItemReader implements ItemReader
   <CreditBill> {
2.     private List<CreditBill> list = new ArrayList<CreditBill>();
3. }
```

```

4.     public CustomCreditBillItemReader(List<CreditBill> list){
5.         this.list = list;
6.     }
7.
8.     @Override
9.     public CreditBill read() throws Exception, UnexpectedInputException,
        ParseException,
10.        NonTransientResourceException {
11.         if (!list.isEmpty()) {
12.             return list.remove(0);
13.         }
14.         return null;
15.     }
16. }

```

其中，9~15 行：每次执行 read 操作返回 list 中的一条记录，并将该记录从 list 中移除；如果 list 中没有记录可以返回，则返回 null 对象。

配置自定义的 ItemReader 非常简单，只需要简单的声明 bean 就可以。配置自定义 ItemReader 的声明参见代码清单 6-97。

完整 Job 配置参见文件：ch06/job/job-custom-itemreader.xml。

代码清单 6-97 配置自定义 ItemReader

```

1. <bean:bean id="customItemRead"
2.     class="com.juxtapose.example.ch06.cust.itemreader.
        CustomCreditBillItemReader">
3. </bean:bean>

```

其中，1~3 行：声明自定义的 ItemReader。

接下来运行自定义 ItemReader，参见代码清单 6-98。

完整代码参见类：test.com.juxtapose.example.ch06.JobLaunchCustomItemReaderTest

代码清单 6-98 执行不可重启的自定义 ItemReader

```

1. List<CreditBill> list = new ArrayList<CreditBill>();
2. list.add(new CreditBill("1","tom",100.00,"2013-2-2 12:00:08","Lu Jia Zui
    road"));
3. list.add(new CreditBill("2","tom",320,"2013-2-3 10:35:21","Lu Jia Zui
    road"));
4. list.add(new CreditBill("3","tom",360.00,"2013-2-11 11:12:38","Longyang
    road"));
5. CustomCreditBillItemReader reader = new CustomCreditBillItemReader(list);
6. Assert.assertEquals("1", reader.read().getAccountID());
7. Assert.assertEquals("2", reader.read().getAccountID());
8. Assert.assertEquals("3", reader.read().getAccountID());
9. Assert.assertNull(reader.read());

```

其中，1~4 行：准备示例数据，完成 list 的初始化对象。

5 行：完成 CustomCreditBillItemReader 的初始化，指定从 list 中读取对象。

6~8 行：连续执行自定义 ItemReader 的 read 操作，获取 list 中的信用卡对账单对象；通过 JUnit 中的断言比较每次读取的对象是否正确。

9 行：第四次执行 read 操作应该返回 null，利用断言 assertNull 来验证读取的内容为 null。

本节实现了自定义的 ItemReader，但本节实现的自定义的 ItemReader 不支持重新启动的能力，导致 Job 作业失败的情况下不能从失败的执行点重新开始读取。6.8.2 章节我们将实现可重启的自定义 ItemReader。

6.8.2 可重启 ItemReader

Spring Batch 框架对 Job 提供了可重启的能力，所有 Spring Batch 框架中提供的 ItemReader 组件均支持可重启的能力。为了支持 ItemReader 的可重启能力，框架定义了接口 ItemStream，所有实现接口 ItemStream 的组件均支持可重启的能力。

ItemStream 接口定义参见代码清单 6-99。

代码清单 6-99 ItemStream 接口定义

```
1. public interface ItemStream {  
2.     void open(ExecutionContext executionContext) throws ItemStreamException;  
3.     void update(ExecutionContext executionContext) throws ItemStreamException;  
4.     void close() throws ItemStreamException;  
5. }
```

其中，2 行：open()操作根据参数 executionContext 打开需要读取资源的 stream，可以根据持久化在执行上下文 executionContext 中的数据重新定位需要读取记录的位置。

3 行：update()操作将 需要持久化的数据存放在执行上下文 executionContext 中。

4 行：close()操作关闭读取的资源。

ItemStream 接口定义了读操作与执行上下文 ExecutionContext 交互的能力。可以将已经读的条数通过该接口存放在执行上下文 ExecutionContext 中（ExecutionContext 中的数据在批处理 commit 的时候会通过 JobRepository 持久化到数据库中），这样到 Job 发生异常重新启动 Job 的时候，读操作可以跳过已经成功读过的数据，继续从上次出错的地点（可以从执行上下文中获取上次成功读的位置）开始读。

接下来我们改造 6.8.1 章节的自定义的 ItemReader，新增实现接口 ItemStream，使其支持可重启的能力。

RestartableCustomCreditBillItemReader 的实现参见代码清单 6-100。

完整代码参见：com.juxtapose.example.ch06.cust.itemreader.RestartableCustomCreditBillItemReader。

代码清单 6-100 RestartableCustomCreditBillItemReader 类定义

```
1. public class RestartableCustomCreditBillItemReader implements
2.     ItemReader<CreditBill>,ItemStream{
3.     private List<CreditBill> list = new ArrayList<CreditBill>();
4.     private int currentLocation = 0;
5.     private static final String CURRENT_LOCATION = "current.location";
6.
7.     public RestartableCustomCreditBillItemReader(List<CreditBill> list){
8.         this.list = list;
9.     }
10.
11.     public CreditBill read() throws Exception, UnexpectedInputException,
12.         ParseException, NonTransientResourceException {
13.         if (!list.isEmpty()) {
14.             return list.get(currentLocation++);
15.         }
16.         return null;
17.     }
18.
19.     public void open(ExecutionContext executionContext)
20.         throws ItemStreamException {
21.         if(executionContext.containsKey(CURRENT_LOCATION)){
22.             currentLocation = new Long(executionContext.getLong(CURRENT_
23.                 LOCATION)).intValue();
24.         }
25.         else{
26.             currentLocation = 0;
27.         }
28.     }
29.
30.     public void update(ExecutionContext executionContext)
31.         throws ItemStreamException {
32.         executionContext.putLong(CURRENT_LOCATION, new Long(currentLocation)
33.             .longValue());
34.     }
35.
36.     public void close() throws ItemStreamException {}
37. }
```

其中，11~17 行：**read** 操作，根据当前的位置 **currentLocation** 从 **list** 中读取数据，当前的位置标识在执行上下文中获取参见 **open** 操作中的代码实现。

19~29 行：**open** 操作，从执行上下文中获取当前读取的位置。

30~34 行：**update** 操作，将当前已经读过的数据位置存放在执行上下文中，通常 **update**

操作在 chunk 的事务提交后会执行一次。

36 行: close 操作, 通常在此处关闭不再需要的资源。

配置自定义的可重启 ItemReader 非常简单, 只需要简单的声明 bean 就可以。配置自定义 ItemReader 的声明参见代码清单 6-101。

完整 Job 配置参见文件: ch06/job/job-custom-itemreader.xml。

代码清单 6-101 配置自定义 ItemReader

```
1. <bean:bean id="restartableCustomItemRead"
2.     class="com.juxtapose.example.ch06.cust.itemreader.
3.         RestartableCustomCreditBillItemReader">
4. </bean:bean>
```

其中, 1~4 行: 声明自定义的可重启的 ItemReader。

接下来运行 RestartableCustomCreditBillItemReader, 参见代码清单 6-102。

完整代码参见类: test.com.juxtapose.example.ch06.JobLaunchCustomItemReaderTest。

代码清单 6-102 执行可重启的自定义 ItemReader

```
1. List<CreditBill> list = new ArrayList<CreditBill>();
2. list.add(new CreditBill("1", "tom", 100.00, "2013-2-2 12:00:08", "Lu Jia Zui
   road"));
3. list.add(new CreditBill("2", "tom", 320, "2013-2-3 10:35:21", "Lu Jia Zui
   road"));
4. list.add(new CreditBill("3", "tom", 360.00, "2013-2-11 11:12:38", "Longyang
   road"));
5.
6. RestartableCustomCreditBillItemReader reader =
7.     new RestartableCustomCreditBillItemReader(list);
8. ExecutionContext executionContext = new ExecutionContext();
9. ((ItemStream) reader).open(executionContext);
10. Assert.assertEquals("1", reader.read().getAccountID());
11. ((ItemStream) reader).update(executionContext);
12.
13. reader = new RestartableCustomCreditBillItemReader(list);
14. ((ItemStream) reader).open(executionContext);
15. Assert.assertEquals("2", reader.read().getAccountID());
```

其中, 1~4 行: 准备示例数据, 完成 list 的初始化对象。

6~7 行: 完成 RestartableCustomCreditBillItemReader 的初始化, 指定从 list 中读取对象。

8 行: 模拟新建执行上下文对象 executionContext。

9 行: 执行自定义 ItemReader 的 open 操作, 从执行上下文中获取当前已读记录的位置。

10 行: 执行一次读操作。

11 行: 执行自定义 ItemReader 的 update 操作, 将当前记录的位置存放在执行上下文中。

13 行: 重新构造一个新的 RestartableCustomCreditBillItemReader 对象, 模拟重启该

Reader。

14 行：使用同一个执行上下文执行 open 操作。

15 行：再次执行读操作，这次会从上次执行的位置来读取数据，因此本次读取的数据是 ID 为 2 的记录。

本示例代码模拟了重新启动读操作的场景，其本质是运行时将游标的位置存放在执行上下文中，执行上下文中的数据在每次事务提交的时候会保存到数据库中；当作业 Job 重新启动的时候从执行上下文中重新获取上次读操作的位置，从正确的位置开始读操作；从而完成了支持重启的功能。

6.9 拦截器

Spring Batch 框架在 ItemReader 执行阶段提供了拦截器，使得在 ItemReader 执行前后能够加入自定义的业务逻辑。ItemReader 执行阶段拦截器接口：org.springframework.batch.core.ItemReadListener<T>。

6.9.1 拦截器接口

接口 ItemReadListener 定义参见代码清单 6-103。

代码清单 6-103 ItemReadListener 接口定义

```
1. public interface ItemReadListener<T> extends StepListener {
2.     void beforeRead();
3.     void afterRead(T item);
4.     void onError(Exception ex);
5. }
```

为 ItemReader 配置拦截器参见代码清单 6-104。

完整配置参见文件：/ch06/job/job-listener.xml。

代码清单 6-104 配置 ItemReader 拦截器

```
1. <job id="itemReadJob">
2.     <step id="itemReadStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="creditItemRead" processor="creditBillProcessor"
5.                 writer="creditItemWriter" commit-interval="2">
6.                 <listeners>
7.                     <listener ref="sysoutItemReadListener"></listener>
8.                     <listener ref="sysoutAnnotationItemReadListener">
9.                         </listener>
10.                 </listeners>
11.             </chunk>
12.         </tasklet>
13.     </step>
14. </job>
```

```

11.         </tasklet>
12.     </step>
13. </job>
14. <bean:bean id="sysoutItemReadListener"
15.     class="com.juxtapose.example.ch06.listener.
        SystemOutItemReadlistener">
16. </bean:bean>
17.
18. <bean:bean id="sysoutAnnotationItemReadListener"
19.     class="com.juxtapose.example.ch06.listener.SystemOutAnnotation">
20. </bean:bean>

```

其中，6~9 行：为作业的读配置 2 个拦截器。

14~16 行：定义拦截器 sysoutItemReadListener，该拦截器实现接口 ItemReadListener。

18~20 行：定义拦截器 sysoutAnnotationItemReadListener，该拦截器通过 Annotation 方式定义。

SystemOutItemReadlistener 的代码参见代码清单 6-105。

类 SystemOutItemReadlistener 完整代码参见：com.juxtapose.example.ch06.listener.SystemOutItemReadlistener。

代码清单 6-105 SystemOutItemReadlistener 类定义

```

1. public class SystemOutItemReadlistener implements ItemReadListener
   <CreditBill> {
2.     public void beforeRead() {
3.         System.out.println("SystemOutItemReadlistener.beforeRead()");
4.     }
5.     public void afterRead(CreditBill item) {
6.         System.out.println("SystemOutItemReadlistener.afterRead()");
7.     }
8.     public void onReadError(Exception ex) {
9.         System.out.println("SystemOutItemReadlistener.onReadError()");
10.    }
11. }

```

6.9.2 拦截器异常

拦截器方法如果抛出异常会影响 Job 的执行，所以在执行自定义的拦截器时候，需要考虑对拦截器发生的异常做处理，避免影响业务。

如果拦截器发生异常，会导致 Job 执行的状态为"FAILED"。

配置了错误拦截器的作业配置参见代码清单 6-106。

完整配置参见文件：/ch06/job/job-listener.xml。

代码清单 6-106 配置错误 ItemReader 拦截器

```

1. <job id="errorItemReadJob">
2.     <step id="errorItemReadStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="creditItemRead"
5.                 processor="creditBillProcessor"
6.                 writer="creditItemWriter" commit-interval="2">
7.                 <listeners>
8.                     <listener ref="errorItemReadListener"></listener>
9.                 </listeners>
10.            </chunk>
11.        </tasklet>
12.    </step>
13.</job>

```

其中，7 行：errorItemReadListener 在 beforeRead 操作中抛出异常，会导致整个作业的失败。

6.9.3 执行顺序

在配置文件中可以配置多个 ItemReadListener，拦截器之间的执行顺序按照 listeners 定义的顺序执行。beforeRead 方法按照 listener 定义的顺序执行，afterRead 方法按照相反的顺序执行。上面示例代码中执行顺序如下：

1. sysoutItemReadListener 拦截器的 before 方法；
2. sysoutAnnotationItemReadListener 拦截器的 before 方法；
3. sysoutAnnotationItemReadListener 拦截器的 after 方法；
4. sysoutItemReadListener 拦截器的 after 方法。

6.9.4 Annotation

Spring Batch 框架提供了 Annotation 机制，可以不实现接口 ItemReadListener，直接通过 Annotation 的机制定义拦截器。为 ItemReadListener 提供的 Annotation 有：

- @BeforeRead；
- @AfterRead；
- @OnReadError。

ItemReadListener 操作说明与 Annotation 定义参见表 6-25。

表 6-25 ItemReadListener 操作说明与 Annotation 定义

操 作	操作说明	Annotation
beforeRead()	在 ItemReader#read()之前执行	@ BeforeRead
afterRead(T item)	在 ItemReader#read()之后执行	@ AfterRead
onReadError(Exception ex)	当 ItemReader#read()抛出异常时候触发该操作	@ OnReadError

使用 Annotation 声明的拦截器的 Spring 配置文件和实现接口 ItemReadListener 的拦截器配置一样，只需要在 listeners 节点中声明即可。

SystemOutAnnotation 的代码参见代码清单 6-107。

类 SystemOutAnnotation 完整代码参见：com.juxtapose.example.ch06.listener.SystemOutAnnotation。

代码清单 6-107 SystemOutAnnotation 类定义

```
1. public class SystemOutAnnotation {
2.     @BeforeRead
3.     public void beforeRead() {
4.         System.out.println("SystemOutAnnotation.beforeRead()");
5.     }
6.
7.     @AfterRead
8.     public void afterRead(CreditBill item) {
9.         System.out.println("SystemOutAnnotation.afterRead()");
10.    }
11.
12.    @OnReadError
13.    public void onReadError(Exception ex) {
14.        System.out.println("SystemOutAnnotation.onReadError()");
15.    }
16. }
```

6.9.5 属性 Merge

Spring Batch 框架提供了多处配置拦截器执行，可以在 chunk 元素节点配置，可以在 tasklet 中配置；框架同样提供了 step 的抽象和继承的功能，可以在父 Step 中定义通用的属性，在子 step 中定义个性化的属性，通过 merge 属性可以定义是覆盖父中的设置，还是和父中的定义合并；chunk 元素中的 listeners 支持 merge 属性。

假设有这样一个场景，所有的 Step 都希望拦截器 sysoutItemReadListener 能够执行，而拦截器 sysoutAnnotationItemReadListener 则由每个具体的 Step 定义是否执行，通过抽象和继承属性可以完成上面的场景。

merge 属性配置代码参见代码清单 6-108。

代码清单 6-108 merge 属性配置

```
1. <job id="mergeChunkJob">
2.     <step id="subChunkStep" parent="abstractParentStep">
3.         <tasklet>
4.             <chunk reader="creditItemRead" processor="creditBillProcessor"
5.                 writer="creditItemWriter" >
```

```

6.         <listeners merge="true">
7.             <listener ref="sysoutAnnotationItemReadListener">
8.                 </listener>
9.             </listeners>
10.        </chunk>
11.    </tasklet>
12. </step>
13.
14. <step id="abstractParentStep" abstract="true">
15.     <tasklet>
16.         <chunk commit-interval="2" >
17.             <listeners>
18.                 <listener ref="sysoutItemReadListener"></listener>
19.             </listeners>
20.         </chunk>
21.     </tasklet>
22. </step>

```

其中，17~18 行：定义抽象作业步 `abstractParentStep` 中的拦截器。

6~8 行：通过 `merge` 属性，可以与父类中的拦截器配置进行合并，表示在 `subChunkStep` 中有两个拦截器会同时工作。

通过 `merge` 属性合并的拦截器的执行顺序如下：首先执行父 `Step` 中定义的拦截器；然后执行子 `Step` 中定义的拦截器。

写数据 ItemWriter

批处理通过 Tasklet 完成具体的任务，chunk 类型的 tasklet 定义了标准的读、处理、写的执行步骤。ItemWriter 是实现写的重要组件，Spring Batch 框架提供了丰富的写基础设施来完成各种数据来源的写入功能，数据来源包括文本文件、Json 格式文件、XML 文件、DB、JMS 消息、写邮件等。

Spring Batch 框架默认提供了丰富的 Writer 实现；如果不能满足需求可以快速方便地实现自定义的数据写入；对于已经存在的持久化服务，框架提供了复用现有服务的能力，避免重复开发。

Spring Batch 框架通常针对大数据量进行处理，同时框架需要将作业处理的状态实时地持久化到数据库中，如果读取一条记录就进行写操作或者状态数据的提交，会大量消耗系统资源，导致批处理框架性能较差。在面向批处理 Chunk 的操作中，可以通过属性 commit-interval 设置 read 多少条记录后进行一次提交。通过设置 commit-interval 的间隔值，减少提交频次，降低资源使用率。属性 commit-interval 的具体使用方式参照 5.3.1 章节。

7.1 ItemWrite

ItemWriter 是 Step 中对资源的写处理阶段，Spring Batch 框架已经提供了各种类型的写实现，包括对文本文件、XML 文件、数据库、JMS 消息、发送邮件等写的处理。写操作与 Chunk Tasklet 的关系参见图 7-1。

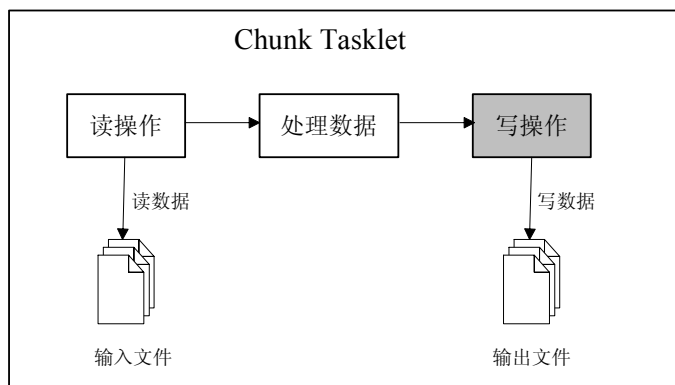


图 7-1 写操作与 Chunk Tasklet 的关系

7.1.1 ItemWriter

所有的写操作需要实现 `org.springframework.batch.item.ItemWriter<T>` 接口。ItemWriter 接口定义参见代码清单 7-1。

代码清单 7-1 ItemWriter 接口定义

```
1. public interface ItemWriter<T> {  
2.     void write(List<? extends T> items) throws Exception;  
3. }
```

其中，2 行：ItemWriter 接口定义了核心作业方法 `write()` 操作，负责将 ItemReader 读入的数据写入指定的资源中；注意这里 `write` 操作的参数是 `List<? extends T> items` 类型的，表示写操作通常会进行批量的写入；每次写入 List 的大小由属性 `commit-interval` 决定。

Job 中典型的配置 ItemWriter 参见代码清单 7-2。

代码清单 7-2 配置 ItemWriter 示例

```
1. <job id="dbReadJob">  
2.     <step id="dbReadStep">  
3.         <tasklet transaction-manager="transactionManager">  
4.             <chunk reader="jdbcParameterItemReader"  
5.                 processor="creditBillProcessor"  
6.                     writer="creditItemWriter" commit-interval="2"></chunk>  
7.             </tasklet>  
8.         </step>  
9.     </job>
```

7.1.2 ItemStream

框架同时提供了另外一个重要的接口 `org.springframework.batch.item.ItemStream`。ItemStream 接口定义了写操作与执行上下文 `ExecutionContext` 交互的能力。可以将已经写的条数通过该接口存放在执行上下文 `ExecutionContext` 中（`ExecutionContext` 中的数据在批处理 `commit` 的时候会通过 `JobRepository` 持久化到数据库中），这样到 Job 发生异常重新启动 Job 的时候，写操作可以跳过已经成功写过的数据，继续从上次出错的地点（可以从执行上下文中获取上次成功写的位置）继续写。

ItemStream 接口定义参见代码清单 7-3。

代码清单 7-3 ItemStream 接口定义

```
1. public interface ItemStream {  
2.     void open(ExecutionContext executionContext) throws ItemStreamException;  
3.     void update(ExecutionContext executionContext) throws ItemStreamException;  
4.     void close() throws ItemStreamException;  
5. }
```

其中，2 行：open()操作根据参数 executionContext 打开需要读取资源的 stream；可以根据持久化在执行上下文 executionContext 中的数据重新定位需要写入记录的位置。

3 行：update()操作将需要持久化的数据存放在执行上下文 executionContext 中。

4 行：close()操作关闭读取的资源。

说明：Spring Batch 框架提供的写组件 FlatFileItemWriter、MultiResourceItemWriter、StaxEventItemWriter 均实现了 ItemStream 接口。

7.1.3 系统写组件

Spring Batch 框架提供的写组件列表参见表 7-1。本章其他章节将重点介绍各种写组件的使用。

表 7-1 Spring Batch 框架提供的写组件

ItemWriter	说 明
FlatFileItemWriter	写 Flat 类型文件
MultiResourceItemWriter	多文件写组件
StaxEventItemWriter	写 XML 类型文件
AmqpItemWriter	写 AMQP 类型消息
ClassifierCompositeItemWriter	根据 Classifier 路由不同的 Item 到特定的 ItemWriter 处理
HibernateItemWriter	基于 Hibernate 方式写数据库
IbatisBatchItemWriter	基于 Ibatis 方式写数据库
ItemWriterAdapter	ItemWriter 适配器，可以复用现有的写服务
JdbcBatchItemWriter	基于 JDBC 方式写数据库
JmsItemWriter	写 JMS 队列
JpaItemWriter	基于 Jpa 方式写数据库
GemfireItemWriter	基于分布式数据库 Gemfire 的写组件
SpELMappingGemfireItemWriter	基于 Spring 表达式语言写分布式数据库 Gemfire 的组件
MimeMessageItemWriter	发送邮件的写组件
MongoItemWriter	基于分布式文件存储的数据库 MongoDB 写组件
Neo4jItemWriter	面向网络的数据库 Neo4j 的读组件
PropertyExtractingDelegatingItemWriter	属性抽取代理写组件；通过调用给定的 Spring Bean 方法执行写入，参数由 Item 中指定的属性字段获取作为参数
RepositoryItemWriter	基于 Spring Data 的写组件
SimpleMailMessageItemWriter	发送邮件的写组件
CompositeItemWriter	条目写的组合模式，支持组装多个 ItemWriter

7.2 Flat 格式文件

Flat 类型文件是一种包含没有相对关系结构的记录的文件。在批处理应用中经常需要处理的文件是简单文本格式文件，这类文件通常没有复杂的关系结构，通常经过分隔符分割，或者定长字段来描述数据格式；稍复杂的文件通过 JSON 的方式定义数据格式。

Spring Batch 框架提供的 `ItemWriter` 本质是将 Java 对象转换为 Flat 文件的记录。

7.2.1 FlatFileItemWriter

`FlatFileItemWriter` 实现 `ItemWriter` 接口，核心作用是将 `ItemReader` 或者 `ItemProcessor` 处理的 Java 对象转换为 Flat 文件中的一行记录，写入到指定的文件中。`FlatFileItemWriter` 通过引用 `LineAggregator`、`FieldExtractor`、`FlatFileHeaderCallback`、`FlatFileFooterCallback` 关键接口实现上面的目的；在 `FlatFileItemWriter` 将 Java 对象转换为文本记录时主要有两步工作，首先根据 `FieldExtractor` 将 Java 对象根据属性抽取出 `Object` 数组，其次使用 `LineAggregator` 将 `Object` 数组转换为 Flat 文本的一行记录，具体参步骤见图 7-2。

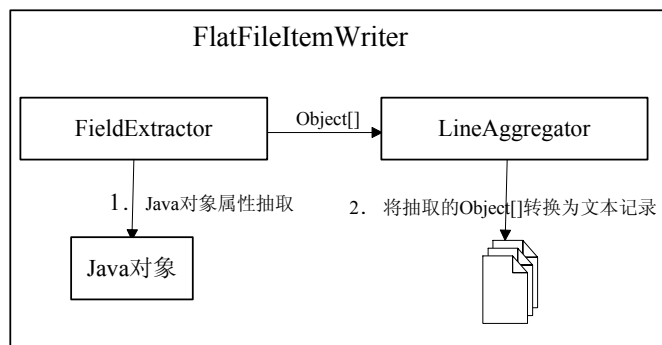


图 7-2 FlatFileItemReader 核心操作步骤

FlatFileItemWriter 结构及关键属性

`FlatFileItemWriter` 与关键接口 `LineAggregator`、`FieldExtractor`、`FlatFileHeaderCallback`、`FlatFileFooterCallback` 之间的类图参见图 7-3。

`FlatFileItemWriter` 引用 `org.springframework.core.io.Resource`，`Resource` 负责提供写入的资源信息，可以是文件，也可以是其他类型的资源；`FlatFileItemWriter` 通过 `LineAggregator` 将 Java 对象转换为文本的一行记录，`LineAggregator` 引用 `FieldExtractor` 后，则负责将 Java 对象根据定义的属性将字段抽取为 `Object` 类型的数组，前者将 `Object` 类型数组转换为文本的一行记录；`FlatFileHeaderCallback` 是写文件头的回调类，在写记录之前会先调用该回调操作，通常在 `open()` 操作中调用该回调操作；`FlatFileFooterCallback` 是写文件尾的回调类，在写记录完成后会调用该回调操作，通常在 `close()` 操作中调用该回调操作。

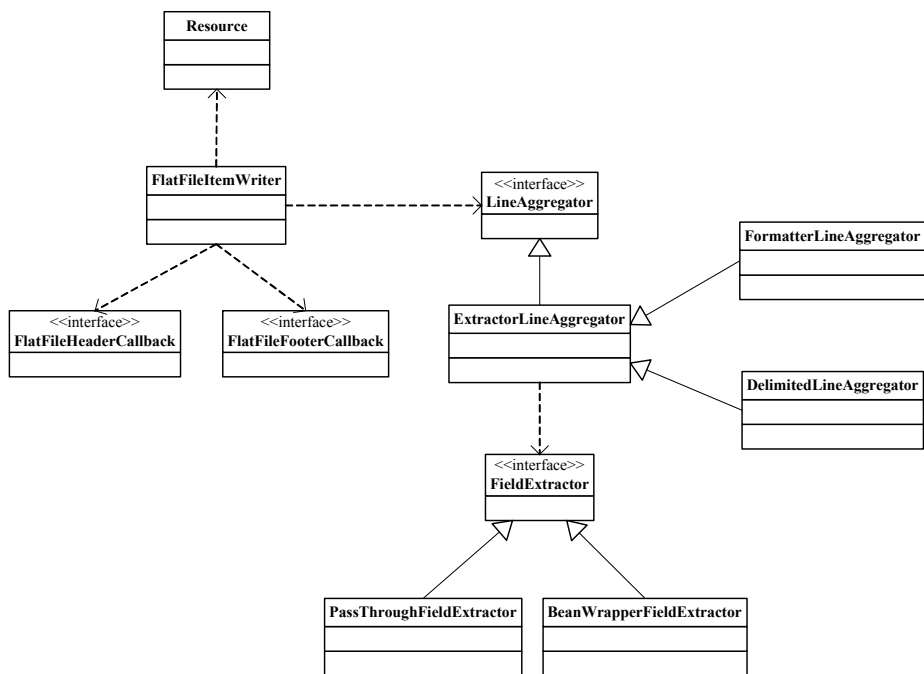


图 7-3 FlatFileItemWriter 类关系

关键接口、类说明参见表 7-2。

表 7-2 FlatFileItemWriter 关键接口、类说明

关 键 类	说 明
Resource	定义写入的文件资源
LineAggregator	将 Item 对象转换为一条文本记录，引用 FieldExtractor 来完成部分工作
FieldExtractor	字段抽取器，负责将 Item 对象根据定义的属性转换为 Object 数组；然后 LineAggregator 将 Object 数组转换为文本的一条记录
FlatFileHeaderCallback	文件头回调类，在 open() 操作中回调该接口，可以在此接口中完成文件头的写入
FlatFileFooterCallback	文件尾回调类，在 close() 操作中回调该接口，可以在此接口中完成文件尾的写入

FlatFileItemWriter 关键属性参见表 7-3。

表 7-3 FlatFileItemWriter 关键属性

FlatFileItemWriter 属性	类 型	说 明
appendAllowed	boolean	写入的文件如果存在，是否采用追加写的方式；如果此属性设置为 true，则属性 shoudDeleteIfExists 自动被设置为 false。 默认值：false

续表

FlatFileItemWriter 属性	类 型	说 明
forceSync	Boolean	是否强制同步写入。 默认值: false
Encoding	String	写入文件的编码类型, 默认值为从环境变量 <code>file.encoding</code> 获取, 如果没有设置则默认为 UTF-8。 默认值: UTF-8
headerCallback	FlatFileHeaderCallback	文件头回调类, 在 <code>open()</code> 操作中回调该接口, 可以在此接口中完成文件头的写入
footerCallback	FlatFileFooterCallback	文件尾回调类, 在 <code>close()</code> 操作中回调该接口, 可以在此接口中完成文件尾的写入
lineAggregator	LineAggregator<T>	将条目对象转为为一行文本
lineSeparator	String	行分隔符。 默认值: 从系统属性 <code>line.separator</code> 中获取
resource	Resource	需要写入的资源文件
saveState	boolean	是否保存写的状态。 默认值: true
shouldDeleteIfEmpty	boolean	没有记录写入文件情况下, 是否删除此文件。 默认值: false
shoudDeleteIfExists	boolean	文件已经存在情况下是否先删除此文件。 默认值: true
transactional	boolean	写是否在事务当中。 默认值: true

配置 FlatFileItemWriter

配置 Flat 格式的文件写相对读取较为简单, 只需要填写必要属性 `resource` 和 `lineAggregator` 即可; 其他属性可以单独设置, 如果不设置则使用上面属性列表中给出的默认值。本例中使用 `FlatFileItemReader` 读取文件 `classpath:ch07/data/flat/credit-card-bill-201310.csv`, 然后写入到文件 `file:target/ch07/flat/outputFile.csv` 中。FlatFileItemWriter 的配置参见代码清单 7-4。

配置 FlatFileItemWriter

完整配置参见文件: `ch07/job/job-flatfile.xml`。

代码清单 7-4 配置 FlatFileItemWriter

```

1.      <bean:bean id="flatFileItemWriter"
2.          class="org.springframework.batch.item.file.FlatFileItemWriter">
3.          <bean:property name="resource" value="file:target/ch07/flat/
           outputFile.csv"/>

```

```

4.      <bean:property name="lineAggregator" ref="lineAggregator"/>
5.      </bean:bean>
6.
7.      <bean:bean id="lineAggregator"
8.          class="org.springframework.batch.item.file.transform.
            DelimitedLineAggregator">
9.          <bean:property name="delimiter" value=","/>
10.         <bean:property name="fieldExtractor">
11.             <bean:bean class="org.springframework.batch.item.
12.                 file.transform.BeanWrapperFieldExtractor">
13.                 <bean:property name="names"
14.                     value="accountID,name,amount,date,address">
15.                 </bean:property>
16.             </bean:bean>
17.         </bean:property>
18.     </bean:bean>

```

其中，3 行：属性 `resource` 定义写入的文件，本例写入文件 `target/ch07/flat/outputFile.csv` 中。

4 行：属性 `lineAggregator` 定义行聚合器对象，`lineAggregator` 负责将 `Item` 对象转换为文本的一行记录，本例使用分隔符的行聚合器 `DelimitedLineAggregator` 来实现该功能。

7~18 行：定义分隔符行聚合器的实现，需要设置两个关键属性 `delimiter` 与 `fieldExtractor`；前者属性 `delimiter` 指定字段间的分隔符，后者属性 `fieldExtractor` 负责根据指定的属性将 `Item` 对象转换为 `Object` 数组。

9 行：定义 `Flat` 格式文件每行中不同字段的分隔符号，本例使用","作为分隔符。

10 行：定义属性字段抽取器，本例使用 `BeanWrapperFieldExtractor` 作为实现，根据 `names` 属性指定的值将 `Item` 对象转换为 `Object` 数组。

使用代码清单 7-5 执行定义的 `flatFileJob`。

完整代码参见：`test.com.juxtapose.example.ch07.JobLaunchFlatFile`。

代码清单 7-5 执行 flatFileJob

```

1. JobLaunchBase.executeJob("ch07/job/job-flatfile.xml", "flatFileJob",
2.     new JobParametersBuilder().addDate("date", new Date()));

```

写入的文件 (`target/ch07/flat/outputFile.csv`) 内容，参见代码清单 7-6。

代码清单 7-6 写入文件 outputFile.csv 文件内容

```

1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road

```

7.2.2 LineAggregator

LineAggregator 负责将 Item 对象转换为一条文本记录，引用 FieldExtractor 来协助完成上面的任务。两者的分工如下：FieldExtractor 负责将 Item 对象转换为 Object 数组；LineAggregator 再将 Object 数组转换为字符串。

接口 `org.springframework.batch.item.file.transform.LineAggregator<T>` 定义参见代码清单 7-7。

代码清单 7-7 LineAggregator 接口定义

```
1. public interface LineAggregator<T> {  
2.     String aggregate(T item);  
3. }
```

aggregate() 操作负责将 Item 对象转换为一条文本记录。

LineAggregator 与 FieldExtractor 的关系参见图 7-4。

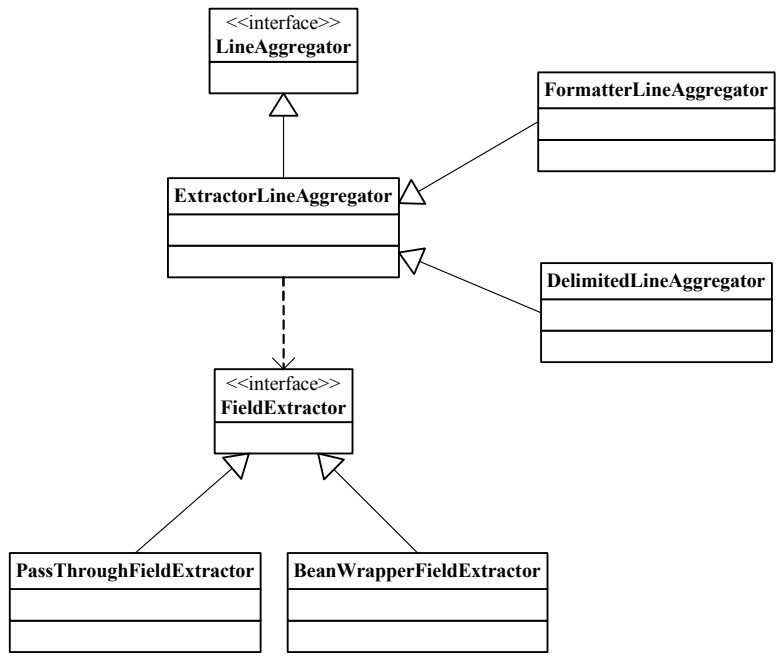


图 7-4 LineAggregator 与 FieldExtractor 类图

LineAggregator 系统实现

Spring Batch 框架中提供了 5 类默认的实现，参见表 7-4，每种实现完成特定的对象转换为一行文本记录的功能，如果默认的 5 类实现不能满足要求，读者可以自行实现 LineAggregator 接口进行转换。

表 7-4 LineAggregator 默认实现

LineAggregator	说 明
ExtractorLineAggregator	抽象类，提供抽象方法 <code>String doAggregate(Object[] fields)</code> ，子类只需要实现将 <code>Object</code> 数组转换为 <code>String</code> 字符串即可。 <code>org.springframework.batch.item.file.transform.ExtractorLineAggregator<T></code>
DelimitedLineAggregator	基于分隔符的行聚合器，将 <code>Item</code> 对象转换为分隔符的一行文本记录；默认的分隔符为","。 <code>org.springframework.batch.item.file.transform.DelimitedLineAggregator<T></code>
FormatterLineAggregator	格式化行聚合器，将 <code>Item</code> 中指定的属性当做参数格式化一个字符串。 <code>org.springframework.batch.item.file.transform.FormatterLineAggregator<T></code>
PassThroughLineAggregator<T>	直接使用 <code>item</code> 的 <code>toString()</code> 操作转换为 <code>String</code> 对象。 <code>org.springframework.batch.item.file.transform.PassThroughLineAggregator<T></code>
RecursiveCollectionLineAggregator<T>	递归集合转换器，将 <code>Collection</code> 对象的每条记录转换为一行，采用默认的换行符号。 <code>org.springframework.batch.item.file.transform.RecursiveCollectionLineAggregator<T></code>

自定义 LineAggregator

接下来我们介绍如何自定义实现 `LineAggregator`。假设账单文件需要每行记录有如下的要求，采用类似分隔符的方式存放属性，且每个属性需要有属性的名字和属性的值，两者之间用等号连接。举例如下，参见代码清单 7-8。

代码清单 7-8 写目标记录格式

```
1. accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
   12:00:08;address=Lu Jia Zui road
```

然后我们创建自定义的类聚合器 `CustomLineAggregator`，将 `Item` 对象转换为上面格式的一行文本。为类 `CustomLineAggregator` 增加两个属性 `delimiter` 和 `names`，前者表示分割的符号，后者 `names` 表示每个字段对应的属性名称。为了实现简单，自定义的类 `CustomLineAggregator` 可以直接继承 `ExtractorLineAggregator`，只需要实现操作 `doAggregate (Object[] fields)` 即可（`doAggregate` 负责将 `Object` 数组转换为 `String` 对象）。

`CustomLineAggregator` 的实现参见代码清单 7-9。

类 `CustomLineAggregator` 完整代码参见：`com.juxtapose.example.ch07.flat.CustomLineAggregator<T>`。

代码清单 7-9 CustomLineAggregator 类定义

```
1. public class CustomLineAggregator<T> extends ExtractorLineAggregator<T> {
2.     private String delimiter = ";";
3.     private String[] names;
```



```

4.
5.     public void setDelimiter(String delimiter) {
6.         this.delimiter = delimiter;
7.     }
8.
9.     public void setNames(String[] names) {
10.        Assert.notNull(names, "Names must be non-null");
11.        this.names = Arrays.asList(names).toArray(new String[names.length]);
12.    }
13.
14.    @Override
15.    protected String doAggregate(Object[] fields) {
16.        List<String> fieldList = new ArrayList<String>();
17.        for (int i = 0; i < names.length; i++) {
18.            fieldList.add(names[i] + "=" + fields[i]);
19.        }
20.        return StringUtils
21.            .arrayToDelimitedString(
22.                fieldList.toArray(new String[fieldList.size()]),
23.                this.delimiter);
24.    }
25. }

```

其中，2 行：定义分隔符属性。

3 行：定义属性名称数组的属性，用于和 Object 中的元素值对应。

15~24 行：将 Object 数组转换为 String 格式的对象。

配置自定义的 CustomLineAggregator，具体配置参见代码清单 7-10。

完整配置参见文件：ch07/job/job-flatfile-custom-aggregator.xml。

代码清单 7-10 配置 CustomLineAggregator

```

1.     <bean:bean id="flatFileItemWriter"
2.         class="org.springframework.batch.item.file.FlatFileItemWriter">
3.         <bean:property name="resource"
4.             value="file:target/ch07/flat/custom-aggregator/
5.                 outputFile.csv"/>
6.         <bean:property name="lineAggregator" ref="customLineAggregator"/>
7.     </bean:bean>
8.
9.     <bean:bean id="customLineAggregator"
10.        class="com.juxtapose.example.ch07.flat.CustomLineAggregator">
11.        <bean:property name="names"
12.            value="accountID,name,amount,date,address"/>

```

```

12.         <bean:property name="fieldExtractor">
13.             <bean:bean class="org.springframework.batch.item.file.
14.                 transform.BeanWrapperFieldExtractor">
15.                 <bean:property name="names"
16.                     value="accountID,name,amount,date,address">
17.                 </bean:property>
18.             </bean:bean>
19.         </bean:property>
20.     </bean:bean>

```

其中，5 行：使用自定义的行聚合器 `customLineAggregator` 定义 `flatFileItemWriter`。

8~20 行：定义 `customLineAggregator`，实现类为 `com.juxtapose.example.ch07.flat.CustomLineAggregator`。属性 `names` 定义字段的名称，属性 `fieldExtractor` 定义使用的字段抽取器。

使用代码清单 7-11 执行定义的 `flatFileJob`。

完整代码参见：`test.com.juxtapose.example.ch07.JobLaunchFlatFileCustomLineAggregator`。

代码清单 7-11 执行 flatFileJob

```

1. JobLaunchBase.executeJob("ch07/job/job-flatfile-custom-aggregator",
2. "flatFileJob",new JobParametersBuilder().addDate("date",new Date()));

```

写入的文件内容参见代码清单 7-12。

代码清单 7-12 写入后的目标文件 outputFile.csv

```

1. accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
   12:00:08;address=Lu Jia Zui road
2. accountID=4047390012345678;name=tom;amount=320.0;date=2013-2-3
   10:35:21;address=Lu Jia Zui road
3. accountID=4047390012345678;name=tom;amount=674.7;date=2013-2-6
   16:26:49;address=South Linyi road
4. accountID=4047390012345678;name=tom;amount=793.2;date=2013-2-9
   15:15:37;address=Longyang road
5. accountID=4047390012345678;name=tom;amount=360.0;date=2013-2-11
   11:12:38;address=Longyang road
6. accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28
   20:34:19;address=Hunan road

```

7.2.3 FieldExtractor

`FieldExtractor` 负责将 `Item` 对象转换为 `Object` 数组，`LineAggregator` 再将 `Object` 数组转换为字符串。

接口 `org.springframework.batch.item.file.transform.FieldExtractor<T>` 定义参见代码清单 7-13。

代码清单 7-13 FieldExtractor 接口定义

```
1. public interface FieldExtractor<T> {
2.     Object[] extract(T item);
3. }
```

extract()操作负责将 Item 对象转换为 Object 类型的数组。
FieldExtractor 与 LineAggregator 的关系参见图 7-5 中的类图。

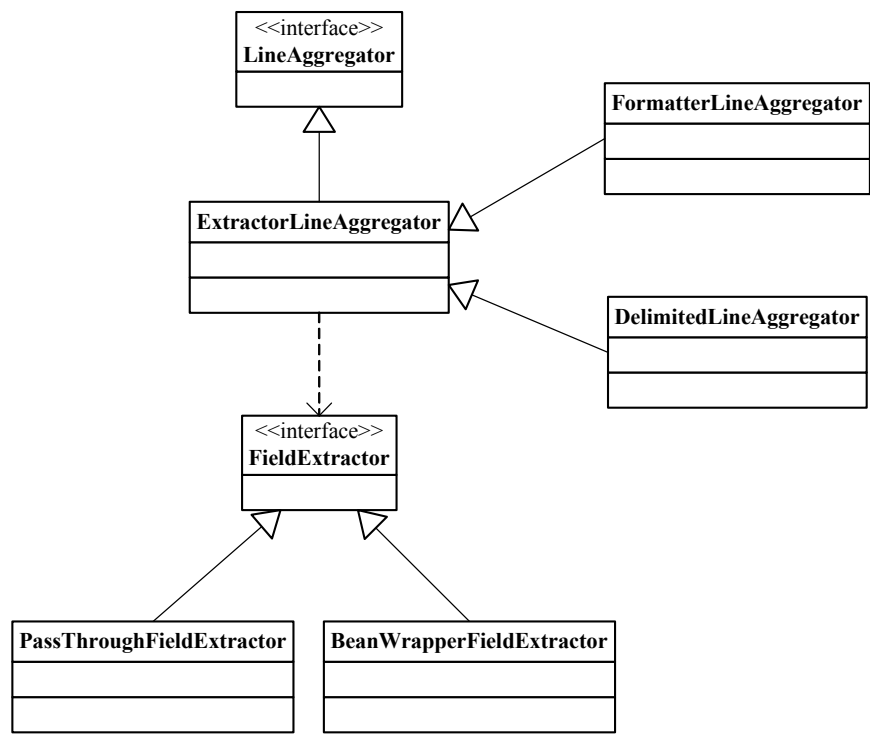


图 7-5 FieldExtractor 与 LineAggregator 的类图

FieldExtractor 系统实现

Spring Batch 框架中提供了 2 类默认的实现，参见表 7-5，每种实现完成特定的对象转换为 Object 类型数组的功能，如果默认的 2 类实现不能满足要求，读者可以自行实现 FieldExtractor 接口进行转换。

表 7-5 FieldExtractor 默认实现

FieldExtractor	说 明
BeanWrapperFieldExtractor	根据给定的属性名，分别调用指定属性的 getter 操作获取属性的值，并组装为 Object 数组直接返回； org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor<T>

FieldExtractor	说 明
PassThroughFieldExtractor	如果 Item 对象是数组类型直接返回; 如果 Item 对象是 Collection 类型, 直接使用 toArray 操作返回; 如果 Item 对象是 Map 类型, 使用 values 返回; 如果 Item 对象是 FieldSet 类型, 使用 getValues 返回; 如果 Item 对象是非上面的类型, 则返回 new Object[] { item }; org.springframework.batch.item.file.transform.PassThroughFieldExtractor<T>

7.2.4 回调操作

在 Flat 类型的写入操作中, 通常在写入文件之前和写入文件之后需要在文本文件中写入额外的内容, 例如注释、写入开始时间、写入完成时间等信息。Spring Batch 框架对 FlatFileItemWriter 增加了写入之前和写入之后的回调操作 FlatFileHeaderCallback 与 FlatFileFooterCallback。

FlatFileHeaderCallback 是写文件头的回调类, 在写记录之前会先调用该回调操作, 通常在 open() 操作中调用该回调操作。

FlatFileFooterCallback 是写文件尾的回调类, 在写记录完成后会调用该回调操作, 通常在 close() 操作中调用该回调操作。

接口声明

接口 FlatFileHeaderCallback 的定义, 参见代码清单 7-14。

接口完整的代码路径: 由 org.springframework.batch.item.file.FlatFileHeaderCallback 定义。

代码清单 7-14 FlatFileHeaderCallback 接口定义

```
1. public interface FlatFileHeaderCallback {
2.     void writeHeader(Writer writer) throws IOException;
3. }
```

writeHeader() 操作在执行真正的文件写入之前调用。

接口 FlatFileFooterCallback 的定义参见代码清单 7-15。

接口完整的代码路径: 由 org.springframework.batch.item.file.FlatFileFooterCallback 定义。

代码清单 7-15 FlatFileFooterCallback 接口定义

```
1. public interface FlatFileFooterCallback {
2.     void writeFooter(Writer writer) throws IOException;
3. }
```

writeFooter() 操作在执行真正的文件写入完成后调用。

FlatFileItemWriter、FlatFileHeaderCallback、FlatFileFooterCallback 三者的序列图参见图 7-6。

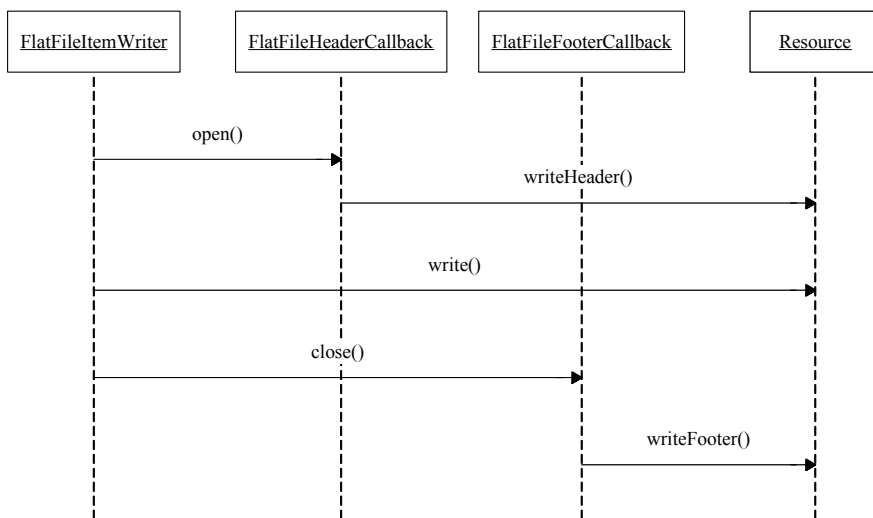


图 7-6 FlatFileItemWriter、FlatFileHeaderCallback、FlatFileFooterCallback 序列图

接口实现

接下来我们实现回调接口，在写入的文件头和文件末尾处写入的开始、结束日期及注释信息。

DefaultFlatFileHeaderCallback 的实现参见代码清单 7-16。

类 DefaultFlatFileHeaderCallback 的完整代码参见：`com.juxtapose.example.ch07.flat.DefaultFlatFileHeaderCallback`。

代码清单 7-16 DefaultFlatFileHeaderCallback 类定义

```

1. public class DefaultFlatFileHeaderCallback implements
   FlatFileHeaderCallback {
2.     public void writeHeader(Writer writer) throws IOException {
3.         writer.write("##credit 201310 begin.");
4.     }
5. }

```

其中，3 行：在文件的头部写入注释“##credit 201310 begin.”。

DefaultFlatFileFooterCallback 的实现参见代码清单 7-17。

类 DefaultFlatFileFooterCallback 完整代码参见：`com.juxtapose.example.ch07.flat.DefaultFlatFileFooterCallback`。

代码清单 7-17 DefaultFlatFileFooterCallback 类定义

```

1. public class DefaultFlatFileFooterCallback implements
   FlatFileFooterCallback {
2.     public void writeFooter(Writer writer) throws IOException {
3.         writer.write("##credit 201310 end.");

```

```
4.     }  
5. }
```

其中，3 行：在文件的尾部写入注释“##credit 201310 end.”。

配置回调

配置 FlatFileItemWriter 的回调操作，具体配置参见代码清单 7-18。

完整配置参见文件：ch07/job/job-flatfile-callback.xml。

代码清单 7-18 配置 FlatFileItemWriter 回调操作

```
1.     <bean:bean id="flatFileComplexItemWriter"  
2.         class="org.springframework.batch.item.file.FlatFileItemWriter">  
3.         <bean:property name="resource"  
4.             value="file:target/ch07/flat/callback/outputFile.csv"/>  
5.         <bean:property name="lineAggregator" ref="lineAggregator"/>  
6.         <bean:property name="headerCallback" ref="headerCallback"/>  
7.         <bean:property name="footerCallback" ref="footerCallback"/>  
8.     </bean:bean>  
9.     <bean:bean id="headerCallback"  
10.         class="com.juxtapose.example.ch07.flat.  
11.             DefaultFlatFileHeaderCallback" />  
12.     <bean:bean id="footerCallback"  
13.         class="com.juxtapose.example.ch07.flat.  
14.             DefaultFlatFileFooterCallback" />
```

其中，6 行：属性 headerCallback 定义文件写入之间的回调操作。

7 行：属性 footerCallback 定义文件写入完成之后的回调操作。

使用代码清单 7-19 执行定义的 flatFileComplexJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchFlatFileCallback。

代码清单 7-19 执行 flatFileComplexJob

```
1. JobLaunchBase.executeJob("ch07/job/job-flatfile-callback.xml",  
2.     "flatFileComplexJob",  
3.     new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容参见代码清单 7-20。

代码清单 7-20 写入后的 outputFile.csv 文件内容

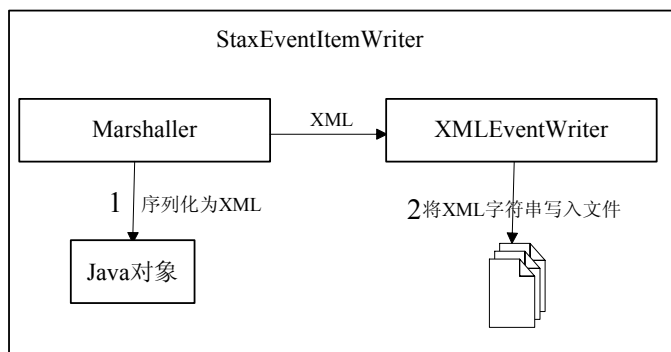
```
1. ##credit 201310 begin.  
2. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road  
3. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road  
4. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road  
5. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road  
6. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road  
7. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road  
8. ##credit 201310 end.
```

其中，1 行：为 DefaultFlatFileHeaderCallback 写入的头信息。

7.3 XML 格式文件

7.3.1 StaxEventItemWriter

StaxEventItemWriter 结构关键属性



StaxEventItemWriter 核心类架构图参见图 7-8。

在实际配置 `StaxEventItemWriter` 时，只需要配置 `Marshaller`、`Resource` 两个属性即可。关键接口、类说明参见表 7-6。

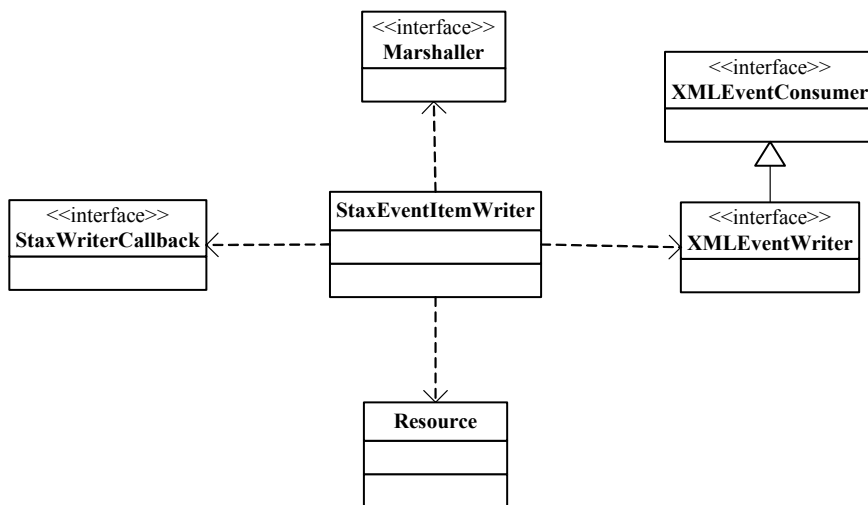


图 7-8 StaxEventItemWriter 核心类架构图

表 7-6 StaxEventItemWriter 关键接口、类说明

关 键 类	说 明
Resource	定义写入的文件资源
Marshaller	序列化类，负责将 Java 对象转换为 XML 片段
XMLEvenWriter	负责提供 StAX 方式对 XML 的写入，基于事件驱动的 XML 处理方式
StaxWriterCallback	写入文件之前、之后的回调类

StaxEventItemWriter 关键属性参见表 7-7。

表 7-7 StaxEventItemWriter 关键属性

StaxEventItemWriter 属性	类 型	说 明
encoding	String	写入 XML 时候的文件编码。 默认值：UTF-8
footerCallback	StaxWriterCallback	写文件尾的回调类，在 close()操作中回调该接口，可以在此接口中完成文件尾的写入
headerCallback	StaxWriterCallback	写文件头的回调类，在 open()操作中回调该接口，可以在此接口中完成文件头的写入
marshaller	Marshaller	Spring OXM 实现类，负责将 Java 对象转换为 XML 内容
overwriteOutput	Boolean	如果文件存在是否覆盖原有的文件。 默认值：true
resource	Resource	需要写入的资源文件

续表

StaxEventItemWriter 属性	类 型	说 明
rootElementAttributes	Map<String,String>	根元素的属性，如果 key 的名字以“xmlns”为前缀表示命名空间
rootTagName	String	根元素的名字。 默认值：root
version	String	指定 XML 的版本号。 默认值：1.0
saveState	Boolean	是否保存写的状态。 默认值：true
transactional	Boolean	写是否在事务当中。 默认值：true
forceSync	Boolean	是否强制同步写入。 默认值：false
shouldDeleteIfEmpty	Boolean	文件已经存在情况下是否先删除此文件。 默认值：true

配置 StaxEventItemWriter

在给出详细配置文件之前，我们首先给出读入的 XML 文件示例（参见代码清单 7-21），通过 XML 的方式描述了信用卡的消费清单。

文件 ch07/data/xml/credit-card-bill-201310.xml。

代码清单 7-21 读入的 XML 文件示例

```
1. <credits>
2.   <credit>
3.     <accountID>4047390012345678</accountID>
4.     <name>tom</name>
5.     <amount>100.00</amount>
6.     <date>2013-2-2 12:00:08</date>
7.     <address>Lu Jia Zui road</address>
8.   </credit>
9.   .....
10. </credits>
```

信用卡账单文件的根节点为 credits，下面为具体的信用卡消费记录节点 credit，在根节点 credits 下面可以有多个信用卡消费记录 credit。

配置 StaxEventItemWriter 代码，参见代码清单 7-22。

完整配置文件参见：ch07/job/job-xml.xml。

代码清单 7-22 配置 StaxEventItemWriter

```
1.      <bean:bean id="xmlWriter"
2.          class="org.springframework.batch.item.xml.StaxEventItemWriter">
3.          <bean:property name="rootTagName" value="juxtapose"/>
4.          <bean:property name="marshaller" ref="creditMarshaller"/>
5.          <bean:property name="resource" value="file:target/ch07/xml/
        credit-card-bill.xml"/>
6.      </bean:bean>
7.      <bean:bean id="creditMarshaller"
8.          class="org.springframework.xml.xstream.XStreamMarshaller">
9.          <bean:property name="aliases">
10.             <util:map id="aliases">
11.                 <bean:entry key="credit"
12.                     value="com.juxtapose.example.ch07.CreditBill"/>
13.             </util:map>
14.          </bean:property>
15.      </bean:bean>
```

其中，3 行：属性 rootTagName 指定写入文件的根节点名称。

4 行：属性 marshaller 指定序列化的组件。

5 行：属性 resource 指定写入的文件路径，为 target/ch07/xml/credit-card-bill.xml。

7~15 行：定义 XML 序列化的组件，此处使用 XStream 的方式进行序列化，使用 Spring OXM 提供的组件 XStreamMarshaller 进行序列化；只需要指定类别名 aliases 属性就可以将特定的 Java 对象转换为 XML 文件，此处使用 com.juxtapose.example.ch07.CreditBill，即将 CreditBill 对象转换为 juxtapose/credit 路径下的 XML 文件片段。

使用代码清单 7-23 执行定义的 xmlFileReadAndWriteJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchXml。

代码清单 7-23 执行 xmlFileReadAndWriteJob

```
1.  JobLaunchBase.executeJob("ch07/job/job-xml.xml",
2.      "xmlFileReadAndWriteJob",
3.      new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容参见代码清单 7-24。

代码清单 7-24 写入后的 credit-card-bill.xml 文件内容

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <juxtapose>
3.      <credit>
4.          <accountID>4047390012345678</accountID>
5.          <name>tom</name>
6.          <amount>100.0</amount>
7.          <date>2013-2-2 12:00:08</date>
```

```

8.      <address>Lu Jia Zui road</address>
9.      </credit>
10.     .....
11. </juxtapose>

```

读者注意，写入后的 XML 的根节点为 juxtapose，是通过属性 rootTagName 进行指定的。

7.3.2 回调操作

在 XML 类型的写入操作中，通常在写入文件之前和写入文件之后需要写入额外的内容，例如注释，写入开始时间，写入完成时间等信息。Spring Batch 框架对 StaxEventItemWriter 增加了写入之前和写入之后的回调操作 StaxWriterCallback，可以通过属性 headerCallback 和 footerCallback 来分别指定。

属性 headerCallback 定义写文件头的回调类，在写记录之前会先调用该回调操作，通常在 open() 操作中调用该回调操作。

属性 footerCallback 定义写文件尾的回调类，在写记录完成后会调用该回调操作，通常在 close() 操作中调用该回调操作。

接口声明

接口 org.springframework.batch.item.xml.StaxWriterCallback 定义，参见代码清单 7-25。

代码清单 7-25 StaxWriterCallback 接口定义

```

1. public interface StaxWriterCallback {
2.     void write(XMLStreamWriter writer) throws IOException;
3. }

```

write() 操作通常在 ItemWriter 的 open() 或者 close() 操作时被触发，负责向指定的 XML 文件写入内容。

StaxEventItemWriter、StaxWriterCallback（可以用属性 headerCallback 和 footerCallback 指定写文件头或者写文件尾）的序列图参见图 7-9。

接口实现

接下来我们实现回调接口，在写入的文件头和文件末尾处写入开始、结束日期及注释信息；本例中在文件头部增加一条 XML 注释信息“<!--credit 201310 begin. -->”，在文件末尾处增加一条 XML 注释信息“<!--Total write count = 3;credit 201310 end. -->”，包含成功写入的记录总条数的信息。

HeaderStaxWriterCallback 的实现参见代码清单 7-26。

类 HeaderStaxWriterCallback 完整代码，参见：com.juxtapose.example.ch07.xml.HeaderStaxWriterCallback。

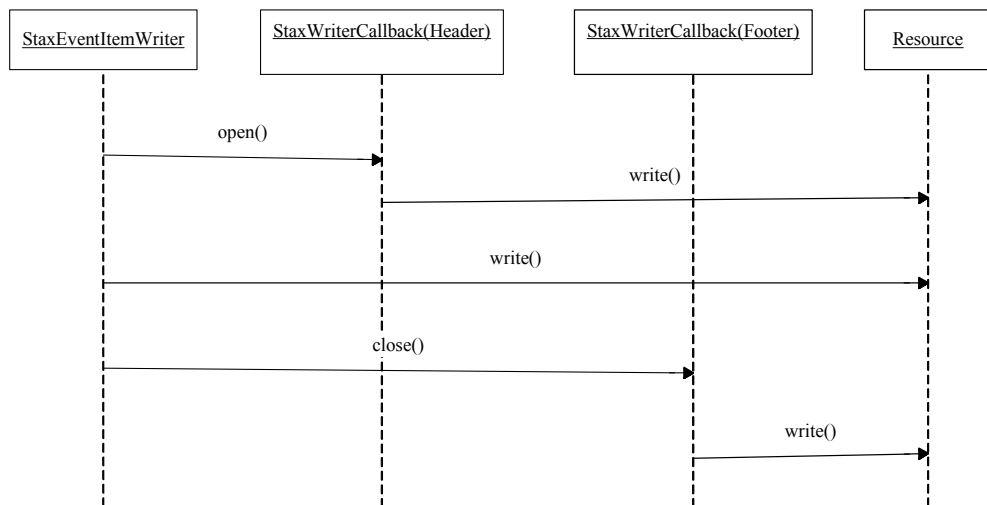


图 7-9 StaxEventItemWriter、StaxWriterCallback 的序列图

代码清单 7-26 HeaderStaxWriterCallback 类定义

```

1. public class HeaderStaxWriterCallback implements StaxWriterCallback {
2.     public void write(XMLStreamWriter writer) throws IOException {
3.         XMLEventFactory factory = XMLEventFactory.newInstance();
4.         try {
5.             writer.add(factory.createComment("credit 201310 begin."));
6.         } catch (XMLStreamException e) {
7.             e.printStackTrace();
8.         }
9.     }
10. }
  
```

其中，5 行：在文件的头部写入注释"credit 201310 begin."。

为了实现获取成功写入的总条目，我们需要在尾部的回调操作中获取作业步的执行上下文，因此我们让 FooterStaxWriterCallback 实现接口 StepExecutionListener，以方便获取作业步的执行上下文。

FooterStaxWriterCallback 的实现参见代码清单 7-27。

类 FooterStaxWriterCallback 完整代码参见：[com.juxtapose.example.ch07.xml.FooterStaxWriterCallback](#)。

代码清单 7-27 FooterStaxWriterCallback 类定义

```

1. public class FooterStaxWriterCallback extends StepExecutionListenerSupport
2.     implements StaxWriterCallback {
3.     public void write(XMLStreamWriter writer) throws IOException {
4.         XMLEventFactory factory = XMLEventFactory.newInstance();
  
```

```

5.         try {
6.             writer.add(factory.createComment("Total write count = "
7.                 + stepExecution.getWriteCount() + ";credit 201310 end."));
8.         } catch (XMLStreamException e) {
9.             e.printStackTrace();
10.        }
11.    }
12.
13.    public void beforeStep(StepExecution stepExecution) {
14.        this.stepExecution = stepExecution;
15.    }
16. }

```

其中，1 行：继承 `StepExecutionListenerSupport` 类，实现了 `StepExecutionListener` 接口，在实现类中获取作业步执行上下文 `StepExecution`。

6~7 行：在文件的尾部写入注释信息，注释包括写入的总行数和写入的日期及结束标志 "end."。

13~15 行：实现 `beforeStep()` 操作，方便在回调实现类 `FooterStaxWriterCallback` 中获取作业步执行上下文 `Step Execution`，通过作业步执行上下文 `Step Execution` 可以方便地获取写入的条目总数等信息。

配置回调

配置 `StaxEventItemWriter` 的回调操作，配置代码参见代码清单 7-28。

完整配置参见文件：`ch07/job/job-flatfile-callback.xml`。

代码清单 7-28 配置 `StaxEventItemWriter` 的回调操作

```

1.     <job id="xmlFileReadAndWriterJob">
2.         <step id="xmlFileReadAndWriterStep">
3.             <tasklet>
4.                 <chunk reader="xmlReader" writer="xmlWriter" commit-
5.                     interval="2"/>
6.                 <listeners>
7.                     <listener ref="footerCallback"/>
8.                 </listeners>
9.             </tasklet>
10.        </step>
11.    </job>
12.    <bean:bean id="xmlWriter"
13.        class="org.springframework.batch.item.xml.StaxEventItemWriter">
14.        <bean:property name="rootTagName" value="juxtapose"/>
15.        <bean:property name="marshaller" ref="creditMarshaller"/>
16.        <bean:property name="resource"

```

```

16.         value="file:target/ch07/xml/callback/credit-card-bill.xml"/>
17.         <bean:property name="headerCallback" ref="headerCallback" />
18.         <bean:property name="footerCallback" ref="footerCallback" />
19.     </bean:bean>
20.     <bean:bean id="headerCallback"
21.         class="com.juxtapose.example.ch07.xml.
22.             HeaderStaxWriterCallback"/>
23.     <bean:bean id="footerCallback"
24.         class="com.juxtapose.example.ch07.xml.
25.             FooterStaxWriterCallback"/>

```

其中,6行:定义作业步执行拦截器 footerCallback,可以方便地在 FooterStaxWriterCallback 中获取作业步执行上下文。

17行: 属性 headerCallback 定义文件写入之间的回调操作。

18行: 属性 footerCallback 定义文件写入完成之后的回调操作。

20~21行: 定义文件写入之间的回调类 com.juxtapose.example.ch07.xml.HeaderStaxWriterCallback。

22~23行: 定义文件写入之后的回调类 com.juxtapose.example.ch07.xml.FooterStaxWriterCallback。

使用代码清单 7-29 执行定义的 xmlFileReadAndWriterJob。

完整代码参见: test.com.juxtapose.example.ch07.JobLaunchXmlCallback。

代码清单 7-29 执行 xmlFileReadAndWriterJob

```

1. JobLaunchBase.executeJob("ch07/job/job-xml-callback.xml",
2.     "xmlFileReadAndWriterJob",
3.     new JobParametersBuilder().addDate("date", new Date()));

```

写入的文件内容,参见代码清单 7-30。

代码清单 7-30 写入后的 credit-card-bill.xml 文件内容

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <juxtapose>
3.     <!-- credit 201310 begin. -->
4.     <credit>
5.         <accountID> 4047390012345678</accountID>
6.         <name> tom</name>
7.         <amount> 100.0</amount>
8.         <date> 2013-2-2 12:00:08</date>
9.         <address> Lu Jia Zui road</address>
10.    </credit>
11.    .....
12.    <!--Total write count = 3;credit 201310 end. -->
13. </juxtapose>

```

其中，3 行：回调实现 `HeaderStaxWriterCallback` 写入的头信息。

12 行：回调实现 `FooterStaxWriterCallback` 写入的尾信息，包含写入记录的总条数。

7.4 写多文件

前面章节完整地介绍了对 Flat 格式、XML 格式文件的写操作，细心的读者会发现上面所有的文件读取基本上是写入单文件。在实际的应用中，批量处理的文件数量非常大，如果全部写入一个文件会导致文件非常大；本节将为读者介绍如何写入多个文件。以信用卡账单为例，在处理信用卡账单时，每个文件存放 100 条记录，超过 100 条之后自动写入一个新的文件，避免每个文件非常大，类似于 Log4j 中的分文件记录日志的能力，参见图 7-10。

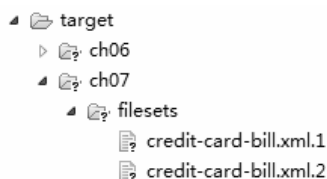


图 7-10 写多文件示例

7.4.1 MultiResourceItemWriter

Spring Batch 框架提供了现有的组件 `MultiResourceItemWriter` 支持对多文件的写入，通过 `MultiResourceItemWriter` 写入批量文件非常简单。`MultiResourceItemWriter` 通过代理的 `ItemWriter` 来读取文件。

MultiResourceItemWriter 结构关键属性

`MultiResourceItemWriter` 组件实现 `ItemWriter`、`ItemStream` 接口，并引用实现了接口 `ResourceAwareItemWriterItemStream` 的读组件，例如 `FlatFileItemWriter`、`StaxEventItemWriter`，通过这些具体的组件完成文件的读取；`ResourceSuffixCreator` 为文件后缀生成器接口，根据给定的 `Resource` 生成不同的文件后缀名，默认情况下使用 `SimpleResourceSuffixCreator` 作为默认实现。

`ResourceSuffixCreator` 接口定义参见代码清单 7-31。

代码清单 7-31 `ResourceSuffixCreator` 接口定义

```
1. public interface ResourceSuffixCreator {  
2.     String getSuffix(int index);  
3. }
```

其中，2 行：根据当前序列 `index` 生成文件后缀名称。

`SimpleResourceSuffixCreator` 的实现非常简单，参见代码清单 7-32。

代码清单 7-32 SimpleResourceSuffixCreator 类定义

```
1. public class SimpleResourceSuffixCreator implements ResourceSuffixCreator {
2.     public String getSuffix(int index) {
3.         return "." + index;
4.     }
5. }
```

其中，3 行：文件名为.+index，其中 index 表示当前的文件序列。

基于 SimpleResourceSuffixCreator 生成的文件名为：

credit-card-bill.xml.1

credit-card-bill.xml.2

.....

MultiResourceItemWriter 核心类结构图，参见图 7-11。

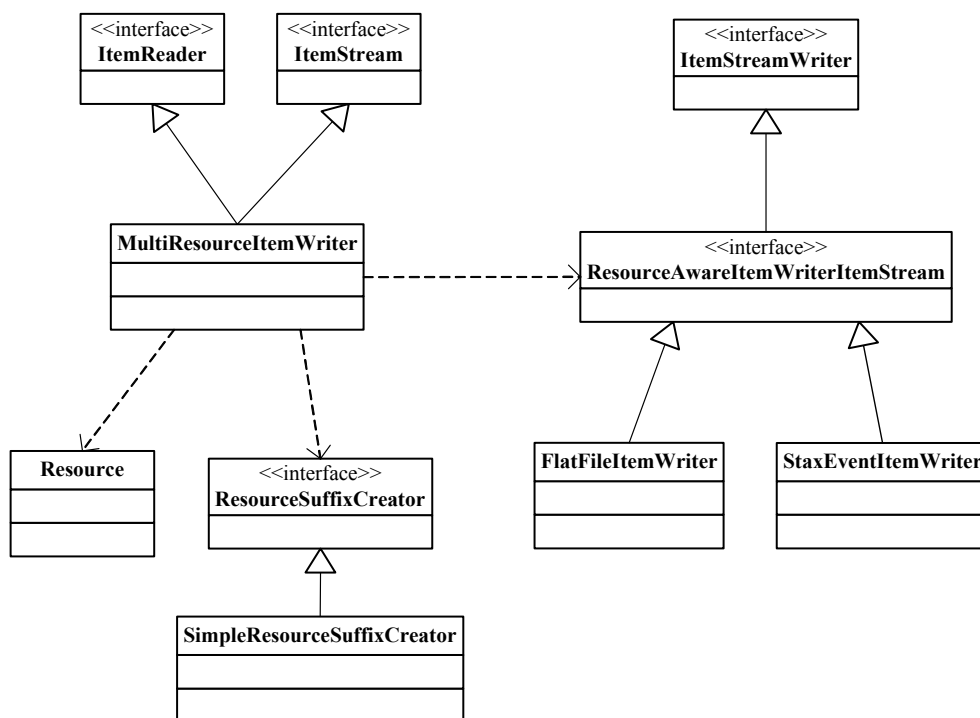


图 7-11 MultiResourceItemWriter 核心类结构图

MultiResourceItemWriter 关键属性参见表 7-8。

配置 MultiResourceItemWriter

本章例子仍然使用上节中的 XML 写入示例，每 2 条记录写入一个新的文件中。配置 MultiResourceItemWriter 的示例参见代码清单 7-33。

完整的配置文件参见：ch07/job/job-filesets.xml。

表 7-8 MultiResourceItemWriter 关键属性

StaxEventItemReader 属性	类 型	说 明
delegate	ResourceAwareItemWriterItemStream	ItemWriter 的代理，将 resources 中定义的文件代理给当前指定的 ItemWriter 进行处理
resource	Resource	需要写入的资源文件名称
suffixCreator	ResourceSuffixCreator	文件后缀生成器，根据给定的文件序列生成文件的后缀名；需实现接口 ResourceSuffixCreator，默认的实现 SimpleResourceSuffixCreator 使用的后缀名为“.index”，index 表示当前的文件序号。 默认值：SimpleResourceSuffixCreator
saveState	boolean	保存状态标识，读取资源时候是否保存当前读取的文件及当前文件读取到条目记录的状态。 默认值:true
itemCountLimitPerResource	int	每个文件可以写入的最大条目，当超过该数值后，会自动写入下一个文件。 默认值：Integer.MAX_VALUE

代码清单 7-33 配置 MultiResourceItemWriter

```
1. <bean:bean id="multiItemWriter"
2.   class="org.springframework.batch.item.file.
   MultiResourceItemWriter" >
3.   <bean:property name="resource"
4.     value="file:target/ch07/credit-card-bill.xml" />
5.   <bean:property name="itemCountLimitPerResource" value="2" />
6.   <bean:property name="delegate" ref="xmlWriter" />
7. </bean:bean>
8. <bean:bean id="xmlWriter"
9.   class="org.springframework.batch.item.xml.StaxEventItemWriter">
10.   <bean:property name="rootTagName" value="juxtapose"/>
11.   <bean:property name="marshaller" ref="creditMarshaller"/>
12. </bean:bean>
```

其中，4 行：属性 resource 指定需要写入的文件资源，其父目录必须存在。

5 行：属性 itemCountLimitPerResource 定义每个文件能写入的最大条目数，本例定义为 2，表示每个文件只能写入 2 条记录。

6 行：属性 delegate 用于指定代理的真正写入的 ItemWriter 的实现类。

8~12 行：定义 xmlWriter 的实现类。

使用代码清单 7-34 执行定义的 filesetsWriteJob。

完整的代码参见：test.com.juxtapose.example.ch07.JobLaunchFileSets。

代码清单 7-34 执行 filesetsWriteJob

```
1. JobLaunchBase.executeJob("ch07/job/job-filesets.xml",  
    "filesetsWriterJob",  
2.     new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容如图 7-12 所示。

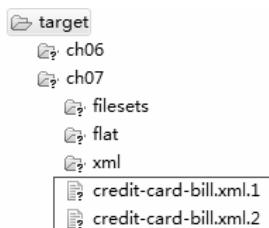


图 7-12 写入的文件内容

说明：MultiResourceItemWriter 中指定的 Resource 其父目录是必须存在的，否则运行期会报如下的错误。修改 multiItemWriter 中的 resource 属性值为："file:target/ch07/notexist/credit-card-bill.xml"，重新执行 JobLaunchFileSets，控制台会出现如下的错误，参见代码清单 7-35。

代码清单 7-35 控制台错误输出

```
java.io.IOException: 系统找不到指定的路径。  
    at java.io.WinNTFileSystem.createFileExclusively(Native Method)  
    at java.io.File.createNewFile(File.java:883)
```

7.4.2 扩展 MultiResourceItemWriter

MultiResourceItemWriter 要求给定的 Resource 的父目录必须存在在，实际使用中可能存在的问题较多，本节我们扩展 MultiResourceItemWriter 实现一个新的支持给定的 Resource，其父目录允许不存在。MultiResourceItemWriter 本身将所有的实现都变成 private 的方式，导致无法通过继承 MultiResourceItemWriter 的方式来扩展，本章节直接使用源代码的方式实现一个新的多文件写的实现类：com.juxtapose.example.ch07.multiresource.ext.ExtMultiResourceItemWriter<T>。

读者可以自行比较 ExtMultiResourceItemWriter 与 MultiResourceItemWriter 的差别，最主要的差别是在 write 方法中创建文件的时候增加了空目录的判断。新的 ExtMultiResourceItemWriter 在 write() 操作中增加了如下的代码片段，具体参见代码清单 7-36。

代码清单 7-36 新的 ExtMultiResourceItemWriter 类定义

```
1. //modify 20130921 begin  
2. if (file.getParent() != null) {  
3.     new File(file.getParent()).mkdirs();  
4. }  
5. //modify 20130921 end
```

配置 ExtMultiResourceItemWriter

本章的例子仍然使用上节中的 XML 写入示例，每 2 条记录写入一个新的文件中。配置 ExtMultiResourceItemWriter 代码参见代码清单 7-37。

完整的配置文件参见：ch07/job/job-filesets.xml。

代码清单 7-37 配置 ExtMultiResourceItemWriter

```
1. <bean:bean id="extMultiItemWriter"
2.     class="com.juxtapose.example.ch07.multiresource.ext.
      ExtMultiResourceItemWriter" >
3.     <bean:property name="resource"
4.         value="file:target/ch07/filesets/credit-card-bill.xml" />
5.     <bean:property name="itemCountLimitPerResource" value="2" />
6.     <bean:property name="delegate" ref="xmlWriter" />
7. </bean:bean>
```

其中，3 行：属性 resource 指定需要写入的文件资源，其父目录可以不存在。

5 行：属性 itemCountLimitPerResource 定义每个文件能写入的最大条目数，本例子定义为 2，表示每个文件只能写入 2 条记录。

6 行：属性 delegate 用于指定代理的真正写入的 ItemWriter 的实现类。

使用代码清单 7-38 执行定义的 extFilesetsWriterJob。

完整的代码参见：test.com.juxtapose.example.ch07.JobLaunchFileSetsExt。

代码清单 7-38 执行 extFilesetsWriterJob

```
1. JobLaunchBase.executeJob("ch07/job/job-filesets.xml",
      "extFilesetsWriterJob",
2.     new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容如图 7-13 所示。

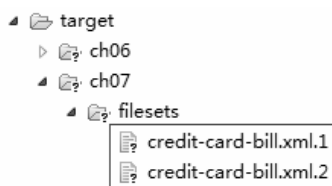


图 7-13 写入后的多文件示例

7.5 写数据库

Spring Batch 框架对写数据库提供了较好的支持，包括基于 JDBC 和 ORM（Object-Relational Mapping）的写入方式。

Spring Batch 框架提供的写数据库组件列表参见表 7-9。

表 7-9 Spring Batch 框架提供的写数据库组件

JdbcBatchItemWriter	基于 JDBC 写数据库组件
HibernateItemWriter	基于 Hibernate 写数据库组件
IbatisBatchItemWriter	基于 Ibatis 写读数据库组件
JpaItemWriter	基于 Jpa 方式写读数据库组件

7.5.1 JdbcBatchItemWriter

Spring Batch 框架提供了对 JDBC 写支持的组件 `JdbcBatchItemWriter`。`JdbcBatchItemWriter` 实现 `ItemWriter` 接口，其核心作用是将 `Item` 对象转换为数据库中的记录。`JdbcBatchItemWriter` 通过引用 `ItemPreparedStatementSetter`、`ItemSqlParameterSourceProvider`、`DataSource` 关键接口实现上面的目的。

`JdbcBatchItemWriter` 对用户屏蔽了数据库访问的操作细节，且提供了批处理的特性，`JdbcBatchItemWriter` 会批量执行一组 SQL 语句来提高性能，而不是逐条执行 SQL 语句；每次批量提交的语句数和 `Chunk` 中定义的提交间隔是一致的。

JdbcBatchItemWriter 结构及关键属性

`JdbcBatchItemWriter` 核心操作步骤，参见图 7-14。

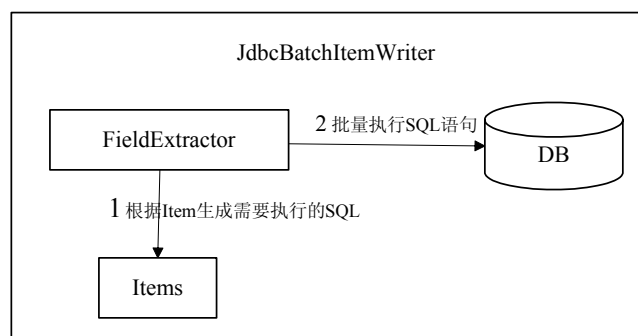


图 7-14 JdbcBatchItemWriter 核心操作步骤

`JdbcBatchItemWriter` 与关键接口 `ItemPreparedStatementSetter`、`ItemSqlParameterSourceProvider`、`DataSource` 之间的类图参见图 7-15。

`JdbcBatchItemWriter` 引用 `javax.sql.DataSource`，`DataSource` 提供数据库信息；`JdbcBatchItemWriter` 通过 `NamedParameterJdbcTemplate` 对数据库进行操作；执行的 SQL 语句如果有“?”的参数，可以通过接口 `ItemPreparedStatementSetter` 进行指定；执行的 SQL 语句如果使用有名字的参数，需要通过接口 `ItemSqlParameterSourceProvider` 进行声明。

`JdbcBatchItemWriter` 关键接口、类说明参见表 7-10。

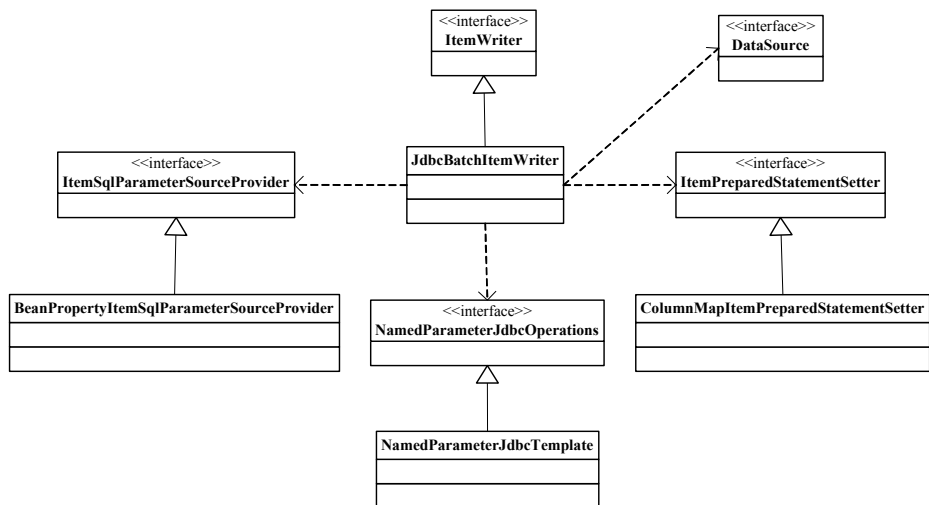


图 7-15 JdbcBatchItemWriter 类关系图

表 7-10 JdbcBatchItemWriter 关键接口、类说明

关 键 类	说 明
DataSource	提供写入数据库的数据源信息
ItemPreparedStatementSetter	为 SQL 语句中有"?"的参数提供赋值接口
ColumnMapItemPreparedStatementSetter	接口 ItemPreparedStatementSetter 的实现类，提供基于列的参数设置
ItemSqlParameterSourceProvider	为 SQL 语句中有命名的参数提供赋值接口
BeanPropertyItemSqlParameterSourceProvider	从给定的 Item 中根据参数名称获取 Item 对应的属性值作为参数
NamedParameterJdbcOperations	JDBCTemplate 操作，提供执行 SQL 的能力

JdbcBatchItemWriter 关键属性参见表 7-11。

表 7-11 JdbcBatchItemWriter 关键属性

JdbcBatchItemWriter 属性	类 型	说 明
dataSource	DataSource	数据源，通过该属性指定使用的数据库信息
sql	String	执行的 SQL 语句
itemSqlParameterSourceProvider	ItemSqlParameterSourceProvider	为 SQL 语句中有命名的参数提供赋值
itemPreparedStatementSetter	ItemPreparedStatementSetter	为 SQL 语句中有"?"的参数提供赋值
assertUpdates	Boolean	当没有修改、删除一条记录时候，是否抛出异常。 默认值：true

配置 JdbcBatchItemWriter

使用 JdbcBatchItemWriter 至少需要配置 dataSource、sql 两个属性；dataSource 指定访问的数据源，sql 用于指定查询的 SQL 语句。

在给出详细配置文件之前，我们首先准备 SQL（使用的为 MySQL 数据库）脚本，示例中使用信用卡账单表，存放信用卡消费记录情况，主要字段包括 ID、ACCOUNTID、NAME、AMOUNT、DATE、ADDRESS；示例从表 t_credit 读取所有的记录，然后通过 JdbcBatchItemWriter 写入到表 t_destcredit 中。

建表脚本参见代码清单 7-39。

完整内容参见文件 ch07/db/create-tables-mysql.sql。

代码清单 7-39 数据库建表脚本

```
1. CREATE TABLE t_destcredit
2.     (ID VARCHAR(10),
3.      ACCOUNTID VARCHAR(20),
4.      NAME VARCHAR(10),
5.      AMOUNT NUMERIC(10,2),
6.      DATE VARCHAR(20),
7.      ADDRESS VARCHAR(128),
8.      primary key (ID)
9.     )
10.  ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

配置 JdbcBatchItemWriter 参见代码清单 7-40。

完整的配置文件参见：ch07/job/job-db-jdbc.xml。

代码清单 7-40 配置 JdbcBatchItemWriter

```
1. <bean:bean id="jdbcItemWriter"
2.     class="org.springframework.batch.item.database.
3.         JdbcBatchItemWriter">
4.     <bean:property name="dataSource" ref="dataSource"/>
5.     <bean:property name="sql"
6.         value="insert into t_destcredit (ID,ACCOUNTID,NAME,AMOUNT,
7.             DATE,ADDRESS)
8.             values (:id,:accountID,:name,:amount,:date,:address)"/>
9.     <bean:property name="itemSqlParameterSourceProvider">
10.         <bean:bean class="org.springframework.batch.item.database.
11.             BeanPropertyItemSqlParameterSourceProvider"/>
12.     </bean:property>
13. </bean:bean>
```

其中，3 行：属性 dataSource 用于配置访问的数据源。

4~6 行：属性 sql 定义需要执行的 SQL 语句，插入 t_destcredit 表，SQL 语句有命名的参数，需要从输入的 Item 中获取参数值赋值给 SQL 语句中的参数，本节使用属性 itemSqlParameterSourceProvider 为 SQL 语句的参数赋值。

7~10 行：属性 `itemSqlParameterSourceProvider` 指定 SQL 语句使用的参数，根据参数的名字到 `Item` 对象中获取对应的属性值。

使用代码清单 7-41 执行定义的 `dbWriteJob`。

完整的代码参见：`test.com.juxtapose.example.ch07.JobLaunchJDBC`。

代码清单 7-41 执行 `dbWriteJob`

```
1. JobLaunchBase.executeJob("ch07/job/job-db-jdbc.xml", "dbWriteJob",
2.     new JobParametersBuilder().addDate("date", new Date()));
```

查看数据库表 `t_destcredit`，有 5 条记录表名 `Job` 执行成功。

为?方式参数赋值

上面我们为命名的参数进行了赋值，接下来我们学习如何为带有“?”的 SQL 语句进行赋值；需要使用接口 `ItemPreparedStatementSetter`。接口 `ItemPreparedStatementSetter` 定义参见代码清单 7-42。

代码清单 7-42 `ItemPreparedStatementSetter` 接口定义

```
1. public interface ItemPreparedStatementSetter<T> {
2.     void setValues(T item, PreparedStatement ps) throws SQLException;
3. }
```

接口 `ItemPreparedStatementSetter` 仅包含一个操作，根据给定的 `Item` 为 `PreparedStatement` 进行设置 SQL 语句中的“?”参数。接下来我们实现该接口 `com.juxtapose.example.ch07.jdbc.DestCreditBillItemPreparedStatementSetter`。

`DestCreditBillItemPreparedStatementSetter` 实现，参见代码清单 7-43。

代码清单 7-43 `DestCreditBillItemPreparedStatementSetter` 类定义

```
1. public class DestCreditBillItemPreparedStatementSetter implements
2.     ItemPreparedStatementSetter<DestinationCreditBill> {
3.
4.     public void setValues(DestinationCreditBill item, PreparedStatement ps)
5.         throws SQLException {
6.         ps.setString(1, item.getId());
7.         ps.setString(2, item.getAccountID());
8.         ps.setString(3, item.getName());
9.         ps.setDouble(4, item.getAmount());
10.        ps.setString(5, item.getDate());
11.        ps.setString(6, item.getAddress());
12.    }
13. }
```

重新配置 `JdbcBatchItemWriter`，具体配置代码参见代码清单 7-44。

完整的配置文件参见：`ch07/job/job-db-jdbc.xml`。

代码清单 7-44 配置带参数的 JdbcBatchItemWriter

```

1.      <bean:bean id="jdbcSetterItemWriter"
2.          class="org.springframework.batch.item.database.
            JdbcBatchItemWriter">
3.          <bean:property name="dataSource" ref="dataSource"/>
4.          <bean:property name="sql" value="insert into t_destcredit
5.              (ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS) values (?, ?, ?, ?, ?, ?)"/>
6.          <bean:property name="itemPreparedStatementSetter">
7.              <bean:bean class="com.juxtapose.example.ch07.jdbc.
8.                  DestCreditBillItemPreparedStatementSetter"/>
9.          </bean:property>
10.     </bean:bean>

```

其中，4~5 行：使用带有"?"的 SQL 语句。

6 行：使用自定义的参数设置类 DestCreditBillItemPreparedStatementSetter 为 SQL 语句的参数赋值。

7.5.2 HibernateItemWriter

对象关系映射（Object Relational Mapping，ORM）是一种为解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，使 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。Spring Batch 框架对 ORM 类型的 Hibernate 提供了写的 ItemWriter。

HibernateItemWriter 实现 ItemWriter 接口，核心作用是将 Item 对象持久化到数据库中。学会使用基于 JDBC 方式的写数据库后，使用 HibernateItemWriter 非常简单。下面我们先看 HibernateItemWriter 的关键支持的属性，参见表 7-12。

HibernateItemWriter 结构及关键属性

表 7-12 HibernateItemWriter 关键属性

HibernateItemWriter 属性	类 型	说 明
clearSession	boolean	在写结束阶段是否将 session 清理掉或者 flush。 默认值：true
hibernateTemplate	HibernateOperations	Hibernate 的模板类，提供基础的操作
sessionFactory	SessionFactory	Hibernate 的 SessionFactory，负责与数据库进行交互

配置 HibernateItemWriter

使用 HibernateItemWriter 仅需要配置属性 hibernateTemplate 属性；hibernateTemplate 属

性提供了对数据库的持久化访问。

在给出详细配置文件之前，我们首先准备实体对象（参见代码清单 7-45）、配置 Hibernate 的 `cfg.xml`（参见代码清单 7-46）。本节示例仍然使用 JDBC 写章节使用的数据库脚本。示例从表 `t_credit` 读取所有的记录，然后通过 `JdbcBatchItemWriter` 写入到表 `t_destcredit` 中。

配置数据实体对象

完整的内容参见类：`com.juxtapose.example.ch07.hibernate.DestinationCreditBill`。

代码清单 7-45 配置数据实体对象

```
1. @Entity
2. @Table(name="t_destcredit")
3. public class DestinationCreditBill {
4.     @Id
5.     @Column(name="ID")
6.     private String id;
7.
8.     @Column(name="ACCOUNTID")
9.     private String accountID = "";    /** 银行卡账户 ID */
10.
11.     @Column(name="NAME")
12.     private String name = "";        /** 持卡人姓名 */
13.
14.     @Column(name="AMOUNT")
15.     private double amount = 0;       /** 消费金额 */
16.
17.     @Column(name="DATE")
18.     private String date;              /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */
19.
20.     @Column(name="ADDRESS")
21.     private String address;           /** 消费场所 */
22.
23.     .....
24. }
```

其中，1 行：`@Entity` 注释声明该类为持久类，将一个 `JavaBean` 类声明为一个实体的数据库表映射类，建议实现序列化。默认情况下，所有的类属性都为映射到数据表的持久性字段。若在类中添加另外属性，而是非映射自数据库的，要用 `Transient` 来注解。

2 行：`@Table(name="t_destcredit")` 持久性映射的表，表名为“`t_destcredit`”。`@Table` 是类一级的注解，定义在 `@Entity` 之下，为实体 `Bean` 映射表，目录和 `schema` 的名字，默认为实体 `Bean` 的类名，不包含包名。

4 行：`@Id`，用于标识数据表的主键。

5 行：`@Column(name="ID")` 表示属性对应的数据库列的字段名。

配置 hibernate 的 cfg 文件

数据实体配置完成后，需要配置 hibernate 的配置文件，用于声明上面的数据实体，hibernate 的 cfg 文件配置参见代码清单 7-46。

完整的文件参见：ch07/cfg/hibernate.cfg.xml。

代码清单 7-46 配置 hibernate 的 cfg 文件

```
1. <hibernate-configuration>
2.     <session-factory>
3.         <mapping class="com.juxtapose.example.ch07.hibernate.CreditBill"/>
4.         <mapping class="com.juxtapose.example.ch07.hibernate.
           DestinationCreditBill"/>
5.     </session-factory>
6. </hibernate-configuration>
```

其中，3 行：声明 Hibernate 中使用的数据实体类 CreditBill。

4 行：声明 Hibernate 中使用的数据实体类 DestinationCreditBill。

配置 HibernateItemWriter

完整的配置文件参见：ch07/job/job-db-hibernate.xml。

下面示例展示了将 Item 数据写入数据库表 t_destcredit 中，具体参见代码清单 7-47。

代码清单 7-47 配置 HibernateItemWriter

```
1. <bean:bean id="hibernateItemWriter"
2.     class="org.springframework.batch.item.database.
           HibernateItemWriter">
3.     <bean:property name="hibernateTemplate" ref="hibernateTemplate" />
4. </bean:bean>
5.
6. <bean:bean id="hibernateTemplate"
7.     class="org.springframework.orm.hibernate3.HibernateTemplate">
8.     <bean:property name="sessionFactory" ref="sessionFactory" />
9. </bean:bean>
10.
11. <bean:bean id="sessionFactory"
12.     class="org.springframework.orm.hibernate3.LocalSession
           FactoryBean">
13.     <bean:property name="dataSource" ref="dataSource"/>
14.     <bean:property name="configurationClass"
15.         value="org.hibernate.cfg.AnnotationConfiguration"/>
16.     <bean:property name="configLocation"
17.         value="classpath:/ch07/cfg/hibernate.cfg.xml"/>
18.     <bean:property name="hibernateProperties">
```

```
19.         <bean:value>
20.             hibernate.dialect=org.hibernate.dialect.MySQLDialect
21.             hibernate.show_sql=true
22.         </bean:value>
23.     </bean:property>
24. </bean:bean>
```

其中，3 行：属性 `hibernateTemplate`，定义 Hibernate 的访问模板，提供持久化操作。

6~9 行：定义 Hibernate 的模板操作，此处使用 Spring 提供的 `org.springframework.orm.hibernate3.HibernateTemplate` 提供访问功能。

11~24 行：声明 hibernate 使用的会话工厂，使用 hibernate 提供的 `LocalSessionFactoryBean`，需要为下面的属性赋值，`dataSource`（数据源）、`configurationClass`（通过注解的方式获取）、`configLocation`（配置文件地址）和 `hibernateProperties`（基本属性信息）。

使用代码清单 7-48 执行定义的 `hibernateWriteJob`。

完整的代码参见：`test.com.juxtapose.example.ch07.JobLaunchHibernate`。

代码清单 7-48 执行 `hibernateWriteJob`

```
1. JobLaunchBase.executeJob("ch07/job/job-db-hibernate.xml",
    "hibernateWriteJob",
2.     new JobParametersBuilder().addDate("date", new Date()));
```

执行完毕，查看数据库表 `t_destcredit`，记录被成功写入表 `t_destcredit`。

7.5.3 IbatisBatchItemWriter

对象关系映射（Object Relational Mapping，ORM）是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

Ibatis 是一个开放源代码的对象关系映射框架，相对于 Hibernate，Ibatis 对 JDBC 进行了更轻量级的对象封装，由于 Iabtis 支持编写 SQL，使得 Ibatis 框架具有更高的灵活性和性能。Spring Batch 框架对 Ibatis 提供了写的 `ItemWriter`。

`IbatisBatchItemWriter` 实现 `ItemWriter` 接口，核心作用是将 `Item` 对象通过 Ibatis 的方式写入到数据库中。学会使用基于 `HibernatePagingItemReader` 的读数据库后，使用 `IbatisBatchItemWriter` 非常简单。下面我们先看 `IbatisBatchItemWriter` 的关键支持的属性，具体内容参见表 7-13。

IbatisBatchItemWriter 结构及关键属性

表 7-13 IbatisBatchItemWriter 关键属性

IbatisBatchItemWriter 属性	类 型	说 明
sqlMapClient	SqlMapClient	用于指定执行的命名 SQL 的配置文件和数据源
statementId	String	命名 SQL 定义的 ID

续表

ibatisBatchItemWriter 属性	类 型	说 明
assertUpdates	boolean	是否断言执行的 SQL 对数据库有记录更新，如果没有更新会抛出异常 EmptyResultDataAccessException。 默认值：true

配置 IbatisBatchItemWriter

使用 IbatisBatchItemWriter 至少需要配置 sqlMapClient、statementId 两个属性；sqlMapClient 用于指定配置的命名 SQL 的文件；statementId 用于指定命名 SQL 文件中定义的 SQL 语句。

在给出详细配置文件之前，我们首先准备配置命名 SQL 的配置文件和命名 SQL 文件。本节示例仍然使用 JDBC 写章节使用的数据库脚本。示例从表 t_credit 读取所有的记录，然后通过 JdbcBatchItemWriter 写入到表 t_destcredit 中。

定义命名 SQL 文件

命名 SQL 框架提供了标准的格式用于定义命名 SQL 文件。本例中的命名 SQL 对表 t_destcredit 进行操作：提供 id 为 “insertDestCredits” 的命名 SQL 用于新增一条信用卡记录。命名 SQL 具体配置参见代码清单 7-49。

完整的文件参见：ch07/ibatis/ibatis-destcredit.xml。

代码清单 7-49 配置命名 SQL

```

1. <sqlMap namespace="DestCredit">
2.   <resultMap id="result" class="com.juxtapose.example.ch07.db.
   DestinationCreditBill">
3.     <result property="id" column="ID" />
4.     <result property="accountID" column="ACCOUNTID" />
5.     <result property="name" column="NAME" />
6.     <result property="amount" column="AMOUNT" />
7.     <result property="date" column="DATE" />
8.     <result property="address" column="ADDRESS" />
9.   </resultMap>
10.   .....
11.
12.   <insert id="insertDestCredits"
13.     parameterClass="com.juxtapose.example.ch07.db.
       DestinationCreditBill">
14.     insert into t_destcredit values(#id#, #accountID#, #name#, #amount#,
       #date#, #address#)
15.   </insert>
16. </sqlMap>

```

其中，1 行：声明当前命名 SQL 文件的命名空间为 “DestCredit”。

2~9 行: 定义命名 SQL 的返回值对象, 将数据库的列与对象 `com.juxtapose.example.ch07.db.DestinationCreditBill` 的属性进行映射, 命名 SQL 执行后会自动将一行转换为指定的对象; 以 `<result property="id" column="ID" />` 举例: 命名 SQL 执行后, 会将表 `t_destcredit` 中的字段 `ID` 映射到对象 `com.juxtapose.example.ch07.db.DestinationCreditBill` 的 `id` 属性上。

12~15 行: 提供 `id` 为 “`insertDestCredits`” 的命名 SQL, 用于向表 `t_destcredit` 中插入一条记录。

配置命名 SQL 的配置文件

命名 SQL 文件定义完成后, 需要在 `Ibatis` 的配置文件中声明, 具体参见代码清单 7-50; 然后可以在 `IbatisBatchItemWriter` 中使用命名 SQL 执行数据库的查询。

完整的文件参见: `ch07/ibatis/ibatis-config.xml`。

代码清单 7-50 配置 `ibatis-config.xml`

```
1. <sqlMapConfig>
2.   <sqlMap resource="ch07/ibatis/ibatis-credit.xml"/>
3.   <sqlMap resource="ch07/ibatis/ibatis-destcredit.xml"/>
4. </sqlMapConfig>
```

其中, 2~3 行: 声明具体的 `Ibatis` 的命名 SQL 文件。

配置 `IbatisBatchItemWriter`

完整的配置文件参见: `ch07/job/job-db-ibatis.xml`。

代码清单 7-51 展示了通过命名 SQL 的方式向数据库表 `t_destcredit` 中插入记录。

代码清单 7-51 配置 `IbatisBatchItemWriter`

```
1. <!-- Ibatis 写数据库 -->
2. <bean:bean id="ibatisItemWriter"
3.   class="org.springframework.batch.item.database.
4.     IbatisBatchItemWriter">
5.   <bean:property name="statementId" value="insertDestCredits" />
6.   <bean:property name="sqlMapClient" ref="sqlMapClient" />
7. </bean:bean>
8. <bean:bean id="sqlMapClient"
9.   class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
10.  <bean:property name="dataSource" ref="dataSource" />
11.  <bean:property name="configLocation"
12.    value="classpath:/ch07/ibatis/ibatis-config.xml" />
13. </bean:bean>
```

其中, 4 行: 属性 `statementId`, 定义访问的命名 SQL 的 ID, 此处访问 ID 为 `insertDestCredits` 的命名 SQL 语句。

5 行: 属性 `sqlMapClient` 定义命名 SQL 的客户端配置文件, 7~12 行给出了该对象的具

体定义，需要为其指定 2 个属性分别是 `dataSource` 和 `configLocation`；`dataSource` 指定使用的数据源，`configLocation` 指定命名 SQL 的配置文件加载地址。

7~12 行：给出了 `sqlMapClient` 对象的具体定义，需要为其指定 2 个属性分别是 `dataSource` 和 `configLocation`；`dataSource` 指定使用的数据源，`configLocation` 指定命名 SQL 的配置文件加载地址。

使用代码清单 7-52，执行定义的 `ibatisWriteJob`。

完整的代码参见：`test.com.juxtapose.example.ch07.JobLaunchIbatis`。

代码清单 7-52 执行 `ibatisWriteJob`

```
1. JobLaunchBase.executeJob("ch07/job/job-db-ibatis.xml",
   "ibatisWriteJob",
2.     new JobParametersBuilder().addDate("date", new Date()));
```

执行完毕，查看数据库表 `t_destcredit`，记录被成功写入表 `t_destcredit`。

7.5.4 JpaItemWriter

对象关系映射（Object Relational Mapping，ORM）是一种为解决面向对象与关系数据库存在的互不匹配现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

JPA（Java Persistence API）是 Sun 官方提出的 Java 持久化规范。JPA 通过 JDK 5.0 注解或 XML 描述对象—关系表的映射关系，并将运行期的实体对象持久化到数据库中；它为 Java 开发人员提供了一种对象/关系映射工具来管理 Java 应用中的关系数据。Spring Batch 框架对 ORM 类型的 JPA 提供了基于写的 `ItemWriter`。

`JpaItemWriter` 实现 `ItemWriter` 接口，核心作用是将 `Item` 对象转换为数据库中的记录。学会使用基于 `JdbcBatchItemWriter` 的数据库写后，使用 `JpaItemWriter` 非常简单。下面我们先看 `JpaItemWriter` 的关键支持的属性，参见表 7-14。

JpaItemWriter 结构及关键属性

表 7-14 JpaItemWriter 关键属性

JpaItemWriter 属性	类 型	说 明
<code>entityManagerFactory</code>	<code>EntityManagerFactory</code>	JPA 提供的实体管理器的工厂类，用于生成实体管理 <code>EntityManager</code> 对象

配置 JpaItemWriter

使用 `JpaItemWriter` 仅需要配置属性 `entityManagerFactory`；`entityManagerFactory` 负责创建 `EntityManager`，后者负责完成对实体的增删改查等。

在给出详细配置文件之前，我们首先准备实体对象（参见代码清单 7-53）、配置 JPA 的

实体映射文件（参见代码清单 7-54）。本节示例仍然使用 7.5.1 章节使用的数据库脚本。

配置数据实体对象

完整的内容参见类 `com.juxtapose.example.ch07.jpa.DestinationCreditBill`。

代码清单 7-53 配置数据实体对象

```
1. @Entity
2. @Table(name="t_destcredit")
3. public class DestinationCreditBill {
4.     @Id
5.     @Column(name="ID")
6.     private String id;
7.
8.     @Column(name="ACCOUNTID")
9.     private String accountID = "";    /** 银行卡账户 ID */
10.
11.    @Column(name="NAME")
12.    private String name = "";        /** 持卡人姓名 */
13.
14.    @Column(name="AMOUNT")
15.    private double amount = 0;       /** 消费金额 */
16.
17.    @Column(name="DATE")
18.    private String date;              /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */
19.
20.    @Column(name="ADDRESS")
21.    private String address;           /** 消费场所 */
22.    .....
23. }
```

其中，1 行：`@Entity` 注释声明该类为持久类，将一个 `JavaBean` 类声明为一个实体的数据库表映射类，建议实现序列化。默认情况下，所有的类属性都为映射到数据表的持久性字段。若在类中，添加另外属性，而非映射来数据库的，要用 `Transient` 来注解。

2 行：`@Table(name="t_destcredit")`持久性映射的表，表名为“`t_destcredit`”。`@Table` 是类一级的注解，定义在`@Entity`之下，为实体 `Bean` 映射表，目录和 `schema` 的名字，默认为实体 `Bean` 的类名，不包含包名。

4 行：`@Id`，用于标识数据表的主键。

5 行：`@Column(name="ID")`表示属性对应的数据库列的字段名。

配置 JPA 的持久化文件

数据实体配置完成后，需要配置 JPA 的实体持久化配置文件，用于声明上面的数据实体。

完整的文件参见：`ch07/jpa/persistence.xml`。

代码清单 7-54 配置 JPA 的实体映射文件

```

1. <persistence>
2.   <persistence-unit name="creditBill" transaction-type="RESOURCE_LOCAL">
3.     <class>com.juxtapose.example.ch07.jpa.CreditBill</class>
4.     <class>com.juxtapose.example.ch07.jpa.DestinationCreditBill
5.       </class>
6.     <exclude-unlisted-classes>true</exclude-unlisted-classes>
7.   </persistence-unit>
8. </persistence>

```

其中，3 行：声明 JPA 中使用的数据实体类 CreditBill，用于 ItemReader 中使用。

4 行：声明 JPA 中使用的数据实体类 DestinationCreditBill，用于 ItemWriter 中使用。

5 行：声明排除所有未在此声明的实体类。

配置 JpaItemWriter

完整配置文件参见：ch07/job/job-db-jpa.xml。

代码清单 7-55 展示了向数据库表 t_destcredit 中插入数据。

代码清单 7-55 配置 JpaItemWriter

```

1. <!-- jpa 写数据库 -->
2. <bean:bean id="jpaItemWriter"
3.   class="org.springframework.batch.item.database.JpaItemWriter">
4.   <bean:property name="entityManagerFactory" ref="entityManager
5.     Factory" />
6. </bean:bean>

```

其中，4 行：属性 entityManagerFactory，定义 JPA 的实体管理器对象，提供访问数据库表的操作；JPA 的实体管理器对象配置参见代码清单 7-56。

代码清单 7-56 配置实体管理器对象

```

1. <bean:bean id="entityManagerFactory"
2.   class="org.springframework.orm.jpa.
3.     LocalContainerEntityManagerFactoryBean">
4.   <bean:property name="dataSource" ref="dataSource" />
5.   <bean:property name="persistenceXmlLocation"
6.     value="classpath:/ch07/jpa/persistence.xml" />
7.   <bean:property name="jpaVendorAdapter">
8.     <bean:bean class="org.springframework.orm.jpa.vendor.
9.       HibernateJpaVendorAdapter">
10.        <bean:property name="showSql" value="true" />
11.      </bean:bean>
12.   </bean:property>
13.   <bean:property name="jpaDialect">

```



```

12.         <bean:bean class="org.springframework.orm.jpa.vendor.
           HibernateJpaDialect" />
13.     </bean:property>
14. </bean:bean>

```

其中，4 行：属性 `persistenceXmlLocation` 指定需要加载的持久化配置文件。

7~11 行：属性 `jpaVendorAdapter` 指定具体的 Provider，此处使用 Hibernate 的实现。

使用代码清单 7-57 执行定义的 `jpaWriteJob`。

完整的代码参见：`test.com.juxtapose.example.ch07.JobLaunchJpa`。

代码清单 7-57 执行 `jpaWriteJob`

```

1. JobLaunchBase.executeJob("ch07/job/job-db-jpa.xml", "jpaWriteJob",
2.     new JobParametersBuilder().addDate("date", new Date())
3.     .addString("id_begin", "1").addString("id_end", "5"));

```

执行完毕，查看数据库表 `t_destcredit`，记录被成功写入表 `t_destcredit`。

7.6 写 JMS 队列

JMS (Java Messaging Service) 是 Java 平台上有关面向消息中间件 (MOM) 的技术规范，它便于消息系统中的 Java 应用程序进行消息交换，并且通过提供标准的产生、发送、接收消息的接口。JMS 是一种与厂商无关的 API，用来访问消息收发系统消息。它类似于 JDBC (Java Database Connectivity)，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。

Spring 的 JMS 抽象框架简化了 JMS API 的使用，并与 JMS 提供者 (比如 IBM 的 WebSphere MQ、开源的 ActiveMQ 等) 平滑地集成。Spring JMS 框架提供了 JMS 访问的模板类 `JmsTemplate`，模板类处理资源的创建和释放，简化了 JMS 的使用。Spring Batch 框架基于 Spring JMS 框架提供了对 JMS 队列写入的 `ItemWriter`。

7.6.1 `JmsItemWriter`

`JmsItemWriter` 实现 `ItemWriter` 接口，它的核心作用是将 `Item` 对象转换为 `Message` 后发送到 JMS 队列。

`JmsItemWriter` 结构关键属性

图 7-16 展示了写 JMS 队列的逻辑架构图，`JmsOperations` 负责将 `Item` 对象转换为 `Message` 消息，并发送到指定的队列中。

`JmsItemWriter` 核心类架构图参见图 7-17。

`JmsItemWriter` 将发送消息全部代理给 `JmsOperations`，`JmsOperations` 将 `Item` 对象转换为 `Message` 对象后发送到 `JmsOperations` 默认指定的队列中。`JmsItemWriter` 关键属性参见表 7-15。

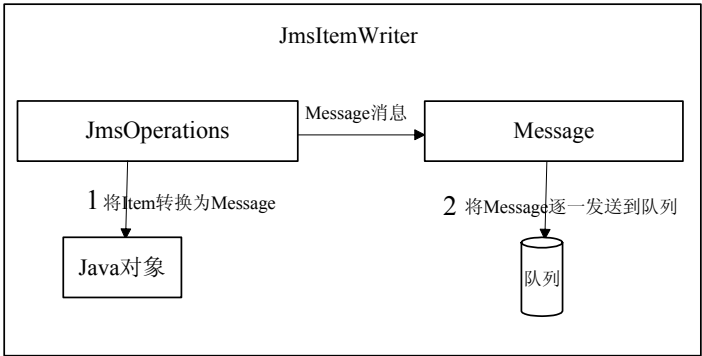


图 7-16 写 JMS 队列的逻辑架构图

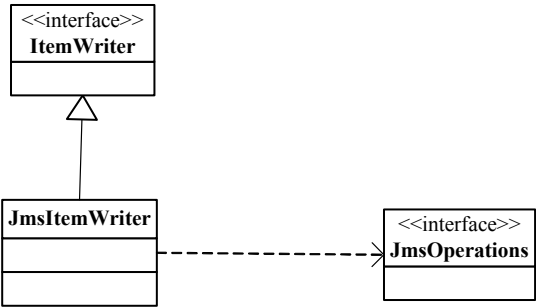


图 7-17 JmsItemWriter 核心类架构图

表 7-15 JmsItemWriter 关键属性

JmsItemWriter 属性	类 型	说 明
jmsTemplate	JmsOperations	消息发送模板

配置 JmsItemWriter

现在假设所有的信用卡账单需要通过 JMS 消息的方式发送到指定的消息队列，需要使用 Spring Batch 框架提供的 JmsItemWriter 来实现该功能。配置 JmsItemWriter 非常简单，只需要指定属性 jmsTemplate 即可，配置代码参见代码清单 7-58。本示例使用 Apache 的 ActiveMQ 作为消息中间件。

完整的配置参见文件 ch07/job/job-jms.xml。

代码清单 7-58 配置 JmsItemWriter

```
1. <bean:bean id="jmsItemWriter"
2.     class="org.springframework.batch.item.jms.JmsItemWriter">
3.     <bean:property name="jmsTemplate" ref="jmsTemplate" />
4. </bean:bean>
```

其中，4 行：属性 jmsTemplate 指定读取 JMS 消息的模板，用于读取指定队列中的消息；

JMS 消息模板的配置参见代码清单 7-59。

代码清单 7-59 JMS 消息模板的配置

```
1. <bean:bean id="jmsTemplate" class="org.springframework.jms.core.  
   JmsTemplate">  
2.     <bean:property name="connectionFactory" ref="jmsFactory"/>  
3.     <bean:property name="defaultDestination" ref="creditDestination"/>  
4.     <bean:property name="receiveTimeout" value="500"/>  
5. </bean:bean>  
6.  
7. <amq:broker useJmx="false" persistent="false">  
8.   <amq:transportConnectors>  
9.     <amq:transportConnector uri="tcp://localhost:61616" />  
10.   </amq:transportConnectors>  
11. </amq:broker>  
12.  
13. <amq:connectionFactory id="jmsFactory" brokerURL="tcp://localhost:  
    61616"/>  
14. <amq:queue id="creditDestination" physicalName="destination.  
    creditBill" />
```

其中，2 行：属性 `connectionFactory` 用于配置 JMS 的连接工厂。

3 行：属性 `defaultDestination` 指定需要读取的目标消息队列。

4 行：属性 `receiveTimeout` 指定读取消息的超时时间。

7~11 行：定义 AMQ 的 broker，提供 JMS 的服务器端，指定对应的 ip 与 port；属性 `persistent` 表示不让消息持久化；属性 `schedulerSupport` 设置为 `false`，禁止掉 AMQ 的延迟发送的功能，可避免因为 AMQ 异常终止后导致无法启动。

13 行：定义 JMS 的连接工厂。

14 行：定义消息存储的队列，AMQ 启动时会自动创建名字为"destination.creditBill"的队列。

增加拦截器，验证发送消息成功。

增加作业步执行拦截器 `StepExecutionListener`，在 `afterStep` 中使用 `JmsTemplate` 读取发送的消息。新实现的拦截器为 `JMSDetectItemWriteListener`，核心代码参见代码清单 7-60。

完整代码参见：`com.juxtapose.example.ch07.jms.JMSDetectItemWriteListener`。

代码清单 7-60 JMSDetectItemWriteListener 类定义

```
1. public class JMSDetectItemWriteListener implements StepExecutionListener {  
2.     private JmsTemplate jmsTemplate;  
3.  
4.     public void beforeStep(StepExecution stepExecution) {}  
5.  
6.     public ExitStatus afterStep(StepExecution stepExecution) {
```

```

7.         int writeCount = 0;
8.         Object obj = jmsTemplate.receiveAndConvert();
9.         while(null != obj){
10.             writeCount++;
11.             CreditBill result = (CreditBill) obj;
12.             System.out.println("Receive from jms queue:"+result);
13.             obj = jmsTemplate.receiveAndConvert();
14.         }
15.         Assert.assertEquals(stepExecution.getWriteCount(), writeCount);
16.         return stepExecution.getExitStatus();
17.     }
18.
19.     public void setJmsTemplate(JmsTemplate jmsTemplate) {
20.         this.jmsTemplate = jmsTemplate;
21.     }
22. }

```

其中，6~17行：使用 `jmsTemplate` 从消息队列中读取消息，打印在控制台，并使用断言确认从队列中读取消息个数与从 `stepExecution` 中获取写入队列的消息个数是一致的。

由于引用了新的命名空间 `amq`，需要在头文件中定义命名空间，参见代码清单 7-61。

代码清单 7-61 声明 `amq` 命名空间

```

1. <bean:beans xmlns="
2.     xmlns:amq="http://activemq.apache.org/schema/core"
3.     xsi:schemaLocation="
4.         http://activemq.apache.org/schema/core
5.         http://activemq.apache.org/schema/core/activemq-core.xsd">
6.     .....
7. </bean:beans>

```

`Job` 及拦截器的配置参见代码清单 7-62。

完整的配置参见文件 `ch07/job/job-jms.xml`。

代码清单 7-62 配置 `Job` 及拦截器

```

1.     <job id="jmsWriteJob">
2.         <step id="jmsWriteStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="flatFileItemReader"
5.                     processor="creditBillProcessor"
6.                     writer="jmsItemWriter" commit-interval="2">
7.                     <listeners>
8.                         <listener ref="jmsDetectItemWriteListener"/>
9.                     </listeners>
10.                 </chunk>
11.             </tasklet>

```

```

11.         </step>
12.     </job>
13.     <bean:bean id="jmsDetectItemWriteListener"
14.         class="com.juxtapose.example.ch07.jms.
            JMSDetectItemWriteListener">
15.         <bean:property name="jmsTemplate" ref="jmsTemplate" />
16.     </bean:bean>

```

其中，7 行：为作业 `jmsWriteJob` 增加作业步拦截器，用于验证消息成功写入队列。

13~16 行：声明作业步拦截器的实现 `JMSDetectItemWriteListener`。

使用代码清单 7-63 执行定义的 `jmsWriteJob`。

完整代码参见：`test.com.juxtapose.example.ch07.JobLaunchJMS`。

代码清单 7-63 执行 `jmsWriteJob`

```

1. JobLaunchBase.executeJob("ch07/job/job-jms.xml", "jmsWriteJob",
2.     new JobParametersBuilder().addDate("date", new Date()));

```

如果没有任何错误产生，表示消息成功写入到指定的队列 `creditDestination` 中，并经过拦截器 `JMSDetectItemWriteListener` 断言后，确认写入消息数与读出的消息个数一致。至此我们学完了 JMS 消息的写入。

7.7 组合写

在 Spring Batch 框架中对于 `Chunk` 只能配置一个 `ItemWriter`，但在有些业务场景中需要将一个 `Item` 同时写到多个不同的资源文件中，即需要写入到多个 `ItemWriter` 中。Spring Batch 框架提供了组合 `ItemWriter`（`CompositeItemWriter`）的模式满足上面的需求。组合 `ItemWriter` 完成的功能参见图 7-18。

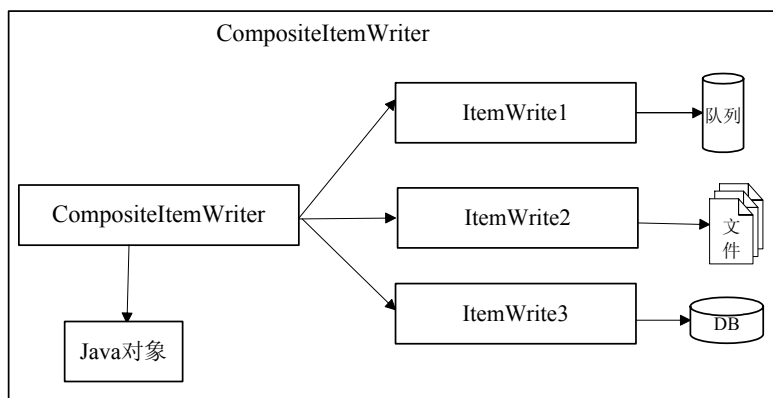


图 7-18 组合 `ItemWriter` 功能示意图

需要注意的是，不同的 `ItemWriter` 写入的记录数是完全相同的。

CompositeItemWriter 结构关键属性

CompositeItemWriter 组件实现 ItemWriter、ItemStream 接口，并引用一组 ItemWriter 实现类，CompositeItemWriter 在进行写入的时候循环调用 ItemWriter 中的实现，将单个 Item 对象写入到不同的 ItemWriter 中。

CompositeItemWriter 核心类结构图参见图 7-19。

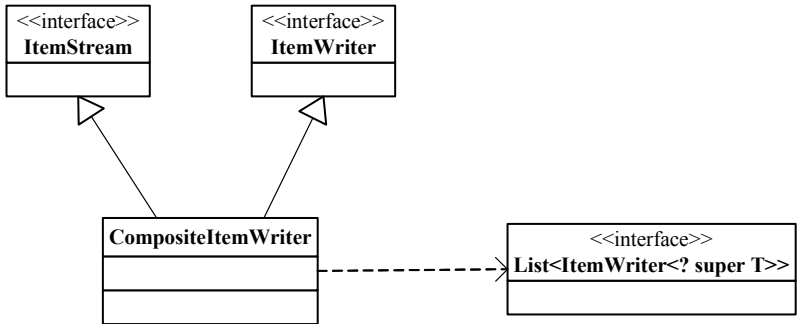


图 7-19 CompositeItemWriter 核心类结构图

CompositeItemWriter 的关键属性参见表 7-16。

表 7-16 CompositeItemWriter 关键属性

CompositeItemWriter 属性	类 型	说 明
delegates	List<ItemWriter<? super T>>	被代理的 ItemWriter 的组合，对于每一个 Item 对象会循环写入到被代理的 ItemWriter 中
ignoreItemStream	boolean	是否忽略被代理的 ItemWriter 实现的 ItemStream 接口的能力。 默认值：false

配置 CompositeItemWriter

在信用卡处理的对象中，需要将信用卡账单写入到不同的 Flat 格式文件中，接下来展示如何使用 CompositeItemWriter 进行组合文件的写入功能。配置 CompositeItemWriter 参见代码清单 7-64。

完整的配置文件参见：ch07/job/job-composite.xml。

代码清单 7-64 配置 CompositeItemWriter

```
1. <bean:bean id="compositeWriter"
2.     class="org.springframework.batch.item.support.
      CompositeItemWriter">
3.     <bean:property name="delegates">
4.         <bean:list>
5.             <bean:ref bean="flatFileItemWriter1" />
```

```

6.         <bean:ref bean="flatFileItemWriter2" />
7.         </bean:list>
8.     </bean:property>
9. </bean:bean>

```

其中，3 行：属性 `delegates` 用于定义需要写入的 `ItemWriter`，本例中将 `Item` 对象写入到两个不同 Flat 格式的文件中。

5 行：属性 `flatFileItemWriter1` 将 `Item` 写入到文件 `file:target/ch07/composite/outputFile1.csv` 中。

6 行：属性 `flatFileItemWriter2` 将 `Item` 写入到文件 `file:target/ch07/composite/outputFile2.csv` 中。

使用代码清单 7-65 执行定义的 `compositeWriteJob`。

完整的代码参见：`test.com.juxtapose.example.ch07.JobLaunchComposite`。

代码清单 7-65 执行 `compositeWriteJob`

```

1. JobLaunchBase.executeJob("ch07/job/job-composite.xml",
    "compositeWriteJob",
2.     new JobParametersBuilder().addDate("date", new Date()));

```

写入的文件内容如图 7-20 所示。

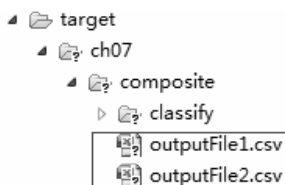


图 7-20 写入后的文件列表

需要注意，`outputFile1.csv` 和 `outputFile2.csv` 的内容是完全一样的，这是组合写模式的特点所在，即将 `Item` 对象写入所有的 `ItemWriter` 中。

7.8 Item 路由 Writer

上节我们学习了如何将一个 `Item` 对象写入到一组 `ItemWriter` 中。另外一些业务场景需要将不同的 `Item` 写入到不同的 `ItemWriter` 中，举例在信用卡账单对象处理过程中需要根据消费的金额将记录写入到不同的文件中：消费金额大于 500 的写入单独的文件，其他小于等于 500 的需要写入另外一个文件中。Spring Batch 框架提供了支持 `Item` 路由写的组件 `ClassifierCompositeItemWriter`。路由 `ItemWriter` 完成的功能如图 7-21 所示。

需要注意的是，不同的 `ItemWriter` 写入的记录数是不同的，根据前面路由的条件来判断写入哪个文件中。

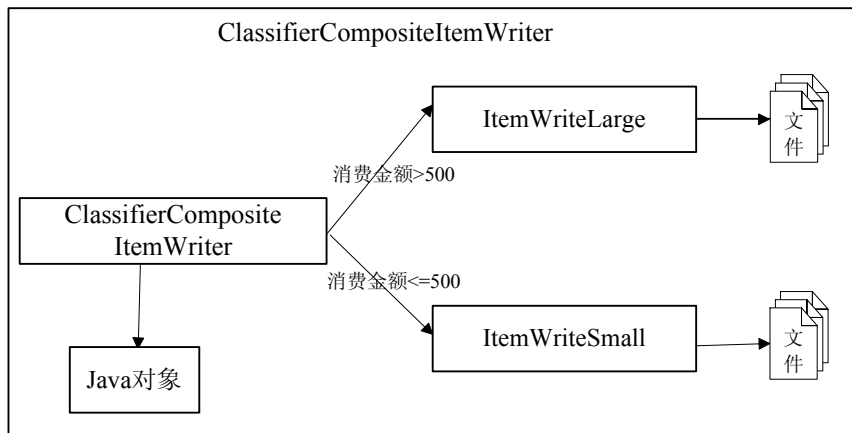


图 7-21 路由 ItemWriter 完成的功能示意图

ClassifierCompositeItemWriter 结构关键属性

ClassifierCompositeItemWriter 组件实现 ItemWriter 接口，Classifier 接口提供路由功能，根据给定的对象返回另外的一个对象；通常可以使用 Classifier 的实现 BackToBackPattern Classifier 来进行路由，BackToBackPatternClassifier 提供根据 Map 的方式支持路由选择。

ClassifierCompositeItemWriter 核心类结构图参见图 7-22。

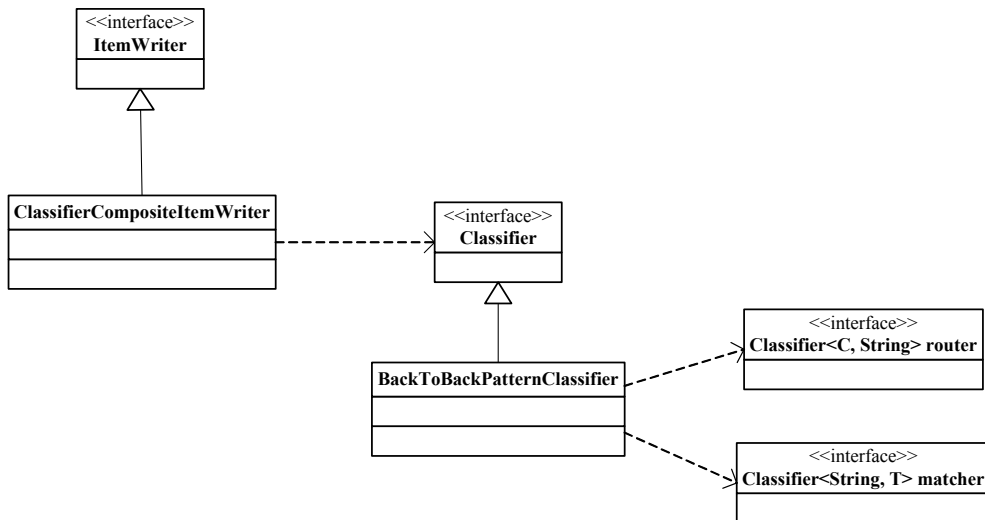


图 7-22 ClassifierCompositeItemWriter 核心类结构图

接口 Classifier 定义参见代码清单 7-66。

接口 Classifier 的完整类路径：org.springframework.classify.Classifier<C, T>。

代码清单 7-66 Classifier 接口定义

```
1. public interface Classifier<C, T> {  
2.     T classify(C classifiable);  
3. }
```

其中，2 行：方法 `classify()` 提供路由功能，输入参数 `C classifiable`，返回 `T` 类型对象。

类 `BackToBackPatternClassifier` 提供了友好的基于 `Map` 路由能力，属性 `router` 根据 `Item` 返回一个字符串，`matcher` 属性提供一个 `Map` 对象，根据 `router` 返回的字符串作为 `key`，从 `Map` 中获取 `ItemWriter` 对象。

接下来我们基于接口 `Classifier` 实现 `router` 的对象 `CreditBillRouterClassifier`，根据信用卡账单对象 `CreditBill` 的金额决定返回的值，如果信用卡账单对象大于 500，则返回"large"；如果信用卡账单没有超过 500，则返回"small"。

`CreditBillRouterClassifier` 实现参见代码清单 7-67。

完整代码参见：`com.juxtapose.example.ch07.classifiercomposite.CreditBillRouterClassifier`。

代码清单 7-67 CreditBillRouterClassifier 类定义

```
1. public class CreditBillRouterClassifier{  
2.     @Classifier  
3.     public String classify(CreditBill classifiable) {  
4.         if(classifiable.getAmount() > 500){  
5.             return "large";  
6.         }else{  
7.             return "small";  
8.         }  
9.     }  
10. }
```

接下来我们使用 `BackToBackPatternClassifier` 展示如何根据 `CreditBill` 对象来路由到不同的 `ItemWrite` 中去。具体配置代码参见代码清单 7-68。

配置 BackToBackPatternClassifier

代码清单 7-68 配置 BackToBackPatternClassifier

```
1. <bean:bean id="backToBackClassifier"  
2.     class="org.springframework.classify.  
3.     BackToBackPatternClassifier" >  
4.     <bean:property name="routerDelegate"  
5.         ref="creditBillRouterClassifier" />  
6.     <bean:property name="matcherMap">  
7.         <bean:map>  
8.             <bean:entry key="large" value-ref="flatFileItemWriterLarge"/>  
9.             <bean:entry key="small" value-ref="flatFileItemWriterSmall"/>  
10.         </bean:map>  
11.     </bean:property>  
12. </bean:bean>
```

```

9.         </bean:property>
10.    </bean:bean>
11.    <bean:bean id="creditBillRouterClassifier"
12.        class="com.juxtapose.example.ch07.classifiercomposite.
            CreditBillRouterClassifier">
13.    </bean:bean>

```

其中，3 行：属性 `routerDelegate` 定义路由代理类，负责根据给定的 `Item` 对象返回路由规则，本处返回 `large` 或者 `small`。

4~9 行：属性 `matcherMap` 用于声明 `Map` 对象，`key` 通常用来定义路由规则，它是属性 `routerDelegate` 的返回值；`value` 通常定义 `ItemWriter` 的实例；通过 `routerDelegate` 与 `matcherMap` 的配合，可以方便地将 `Item` 路由到不同的 `ItemWrite` 进行处理。

`ClassifierCompositeItemWriter` 关键属性参见表 7-17。

表 7-17 `ClassifierCompositeItemWriter` 关键属性

ClassifierCompositeItemWriter 属性	类 型	说 明
classifier	Classifier<T, ItemWriter<? super T>>	根据给定的 <code>Item</code> 对象，选择不同的 <code>ItemWrite</code> 进行处理

配置 ClassifierCompositeItemWriter

接下来我们展示使用 `ClassifierCompositeItemWriter` 将不同的信用卡账单记录存入到不同的文件中。具体配置参见代码清单 7-69。

配置 ClassifierCompositeItemWriter

完整配置文件参见：ch07/job/job-composite-classify.xml。

代码清单 7-69 配置 `ClassifierCompositeItemWriter`

```

1.    <bean:bean id="classifierCompositeWriter"
2.        class="org.springframework.batch.item.support.
            ClassifierCompositeItemWriter">
3.        <bean:property name="classifier" ref="backToBackClassifier">
4.        </bean:property>
5.    </bean:bean>
6.    <bean:bean id="backToBackClassifier"
7.        class="org.springframework.classify.BackToBackPatternClassifier">
8.        <bean:property name="routerDelegate"
9.            ref="creditBillRouterClassifier" />
10.        <bean:property name="matcherMap">
11.            <bean:map>
12.                <bean:entry key="large" value-ref="flatFileItemWriterLarge"/>
13.                <bean:entry key="small" value-ref="flatFileItemWriterSmall"/>

```

```

13.         </bean:map>
14.     </bean:property>
15. </bean:bean>

```

其中,3 行:属性 classifier 用于定义路由规则的实现,将 Item 对象路由到不同的 ItemWrite 中处理。

11 行:属性 flatFileItemWriterLarge 将 Item 写入到文件 file:target/ch07/composite/classify/outputFile_Large.csv。

12 行:属性 flatFileItemWriter2 将 Item 写入到文件 file:target/ch07/composite/classify/outputFile_Small.csv。

使用代码清单 7-70 执行定义的 classifierCompositeWriteJob。

完整代码参见: test.com.juxtapose.example.ch07.JobLaunchClassifierComposite。

代码清单 7-70 执行 classifierCompositeWriteJob

```

1. JobLaunchBase.executeJob("ch07/job/job-composite-classify.xml",
2.     "classifierCompositeWriteJob",new JobParametersBuilder().addDate
    ("date", new Date()));

```

文件 outputFile_Large.csv 写入信用卡账单记录如下(金额全部大于 500),具体参见代码清单 7-71。

代码清单 7-71 文件 outputFile_Large.csv 内容清单

```

1. 4047390012345678,tom,674.7,2013-2-6 16:26:49, South Linyi road
2. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road
3. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road

```

文件 outputFile_Small.csv 写入信用卡账单记录如下(金额全部小于 500),具体参见代码清单 7-72。

代码清单 7-72 文件 outputFile_Small.csv 内容清单

```

1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road

```

7.9 发送邮件

Spring Batch 框架提供了发送邮件的功能,使用 org.springframework.batch.item.mail.SimpleMailMessageItemWriter 可以轻松地完成邮件发送的功能。

7.9.1 SimpleMailMessageItemWriter

本节我们使用的信用卡账单需要每期都发送到用户指定的邮箱中,我们使用 SimpleMailMessageItemWriter 展示如何发送邮件功能。

SimpleMailMessageItemWriter 结构关键属性

图 7-23 展示了发送邮件的逻辑架构图，ItemProcessor 负责将 Item 对象转换为 SimpleMailMessage 消息；MailSender 将生成的消息发送到具体的邮箱中。

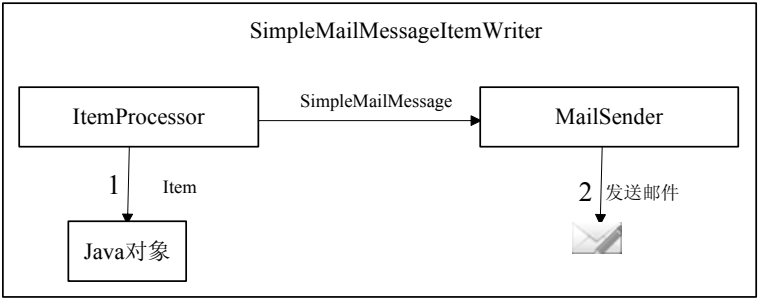


图 7-23 发送邮件的逻辑架构图

SimpleMailMessageItemWriter 关键属性参见表 7-17。

表 7-18 SimpleMailMessageItemWriter 关键属性

SimpleMailMessageItemWriter 属性	类 型	说 明
mailSender	MailSender	邮件发送工具类，负责具体的邮件发送
mailErrorHandler	MailErrorHandler	邮件发送失败情况下的消息处理类。 默认值： org.springframework.batch.item.mail.DefaultMailErrorHandler

通常情况下只需要设置属性 mailSender 即可。需要注意 SimpleMailMessageItemWriter 中 write 操作处理的参数类型为 SimpleMailMessage，因此在调用 SimpleMailMessageItemWriter 之前需要将数据准备正确。

配置 SimpleMailMessageItemWriter

本例中将从 Flat 文件中读取的账单信息，发送到指定的信箱中。首先准备 SimpleMailMessage，我们新实现一个 ItemProcessor，实现类为 MailItemProcessor，实现代码参见代码清单 7-73，用于处理从 Flat 文件中读取的 CreditBill 信息，转换为邮件发送需要的 SimpleMailMessage 对象。

实现 MailItemProcessor

完整代码参见：com.juxtapose.example.ch07.mail.MailItemProcessor。

代码清单 7-73 MailItemProcessor 类定义

```
1. public class MailItemProcessor implements ItemProcessor<CreditBill,  
   SimpleMailMessage> {  
2.     @Override
```

```

3.     public SimpleMailMessage process(CreditBill item) throws Exception {
4.         SimpleMailMessage msg = new SimpleMailMessage();
5.         msg.setFrom("springbatchexample@163.com");
6.         msg.setTo("springbatchexample@163.com");
7.         msg.setSubject("Credit detail " +
8.             new SimpleDateFormat("yyyy 年 MM 月 dd 日 hh 时 mm 分 ss 秒").
9.             format(Calendar.getInstance().getTime()));
10.        msg.setText("Credit details: " + item.toString());
11.        return msg;
12.    }
13. }

```

其中，3～12 行：处理传入的信用卡账单对象，生成需要发送的邮件信息 SimpleMailMessage。

5 行：指定邮件的发件人为“springbatchexample@163.com”。

6 行：指定邮件的收件人为“springbatchexample@163.com”。

7～9 行：指定邮件的主题信息。

10 行：定义邮件的正文。

配置 SimpleMailMessageItemWriter

配置代码参见代码清单 7-74。

完整配置文件参见：ch07/job/job-java-mail.xml。

代码清单 7-74 配置 SimpleMailMessageItemWriter

```

1.     <context:property-placeholder
2.         location="classpath:/ch07/properties/batch-mail.properties"
3.         ignore-unresolvable="true"/>
4.
5.     <bean:bean id="mailItemWriter"
6.         class="org.springframework.batch.item.mail.
7.             SimpleMailMessageItemWriter">
8.         <bean:property name="mailSender" ref="javaMailSender" />
9.     </bean:bean>
10.
11.    <bean:bean id="javaMailSender"
12.        class="org.springframework.mail.javamail.JavaMailSenderImpl">
13.        <bean:property name="host" value="${mail.smtp.host}" />
14.        <bean:property name="username" value="${mail.smtp.username}" />
15.        <bean:property name="password" value="${mail.smtp.password}" />
16.        <bean:property name="javaMailProperties">
17.            <bean:props>
18.                <bean:prop key="mail.smtp.auth">${mail.smtp.auth}</bean:prop>

```

```

18.         <bean:prop key="mail.smtp.timeout">${mail.smtp.timeout}
           </bean:prop>
19.     </bean:props>
20. </bean:property>
21. </bean:bean>

```

其中, 1~3 行: 指定发送邮件需要的属性配置文件, 该文件定义了后续需要用到的变量, 包括 12、13、14、17、18 行中的 `${}` 中的内容。

4~8 行: 定义邮件发送类的实现, 只需要定义属性 `mailSender`。

10~21 行: 定义邮件发送的具体实现类 `org.springframework.mail.javamail.JavaMailSenderImpl`。

12 行: 属性 `host` 定义发送邮件的主机地址, 本例使用 163 邮箱, 设置为 `"smtp.163.com"`。

13 行: 属性 `username` 声明发件人信息, 账户为 `"springbatchexample"`。

14 行: 属性 `password` 声明发件人的密码, 密码为 `"springbatch"`。

17 行: 属性 `mail.smtp.auth` 定义是否采用安全策略, `true` 表示需要输入用户名口令信息。

18 行: 属性 `mail.smtp.timeout` 定义超时时间, 超时时间为 25000 毫秒。

`batch-mail.properties` 的详细信息参见代码清单 7-75。

代码清单 7-75 `batch-mail.properties` 的详细信息

```

1. mail.smtp.host=smtp.163.com
2. mail.smtp.username=springbatchexample
3. mail.smtp.password=springbatch
4. mail.smtp.auth=true
5. mail.smtp.timeout=25000

```

使用代码清单 7-76 执行定义的 `javaMailJob`。

完整代码参见: `test.com.juxtapose.example.ch07.JobLaunchJavaMail`。

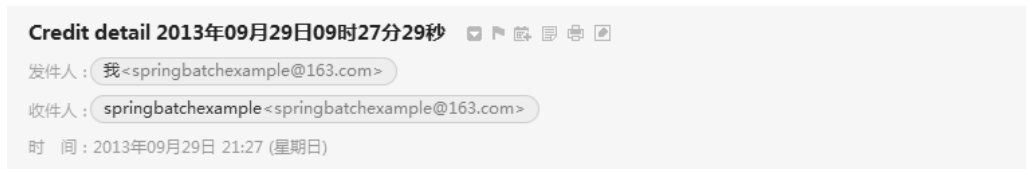
代码清单 7-76 执行 `javaMailJob`

```

1. JobLaunchBase.executeJob("ch07/job/job-java-mail.xml", "javaMailJob",
2.     new JobParametersBuilder().addDate("date", new Date()));

```

执行成功后, 读者可以登录到 163 的邮箱 (账户: `springbatchexample`, 密码: `springbatch`), 可以到成功发送的邮件信息如图 7-24。



Credit details: accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28 20:34:19;address=Hunan road

图 7-24 发送邮件的截图

7.10 服务复用

复用现有的企业资产和服务是提高企业应用开发的快捷手段，Spring Batch 框架的写组件提供了复用现有服务的能力，利用 Spring Batch 框架提供的 `ItemWriterAdapter`、`PropertyExtractingDelegatingItemWriter` 可以方便地复用业务服务、Spring Bean、EJB 或者其他远程服务。`ItemWriterAdapter` 代理的现有服务需要能够处理 `Item` 对象；`PropertyExtractingDelegatingItemWriter` 代理的服务支持更复杂的参数，参数可以根据指定的属性从 `Item` 中抽取，接下来我们介绍如何使用已经存在的服务。

7.10.1 ItemWriterAdapter

ItemWriterAdapter 结构关键属性

`ItemWriterAdapter` 持有服务对象，并调用指定的操作来完成 `ItemWriter` 中定义的 `write` 功能。需要注意的是：已经存在的服务需要能够直接处理 `Item` 对象，即参数必须是 `Item` 的具体类型。如果是更复杂的参数可以使用 `PropertyExtractingDelegatingItemWriter` 或者自己实现新的 `ItemWriter` 来完成从 `Item` 对象到现有服务参数的转变功能。

`ItemWriterAdapter` 和现有服务之间的关系参见图 7-25。

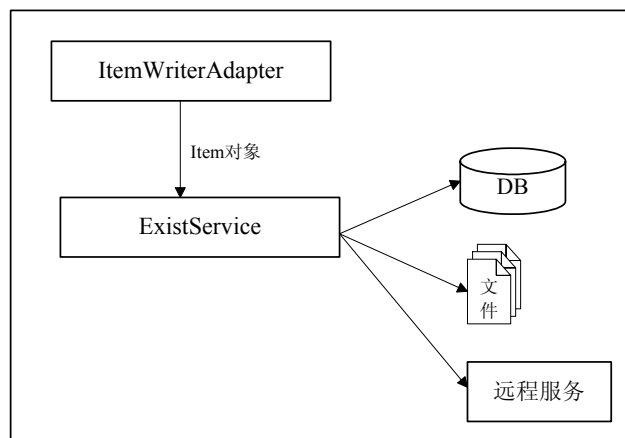


图 7-25 `ItemWriterAdapter` 和现有服务之间的关系

`ItemWriterAdapter` 类关系图参见图 7-26。

`ItemWriterAdapter` 实现接口 `ItemWriter` 并继承 `AbstractMethodInvokingDelegator`，后者提供了调用代理服务的一系列方法；`ExistService` 表示现存的服务。

ItemWriterAdapter 关键属性

表 7-19 给出了 `ItemWriterAdapter` 的关键属性列表。在配置 `ItemWriterAdapter` 时候只需

要指定上面的前两个属性即可，参数 arguments 默认情况下将每次处理的 Item 对象作为参数传入。

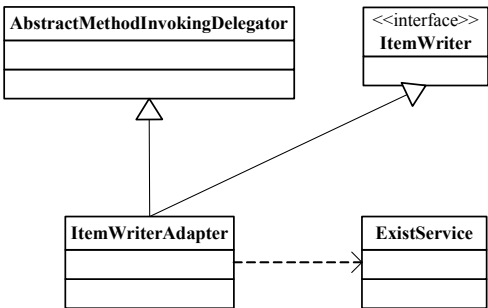


图 7-26 ItemWriterAdapter 类关系图

表 7-19 ItemWriterAdapter 关键属性

ItemWriterAdapter 属性	类 型	说 明
targetObject	Object	需要调用的目标服务对象
targetMethod	String	需要调用的目标操作名称
arguments	Object[]	需要调用的操作参数 默认不需要传递该参数，默认情况下将每次处理的 Item 对象作为参数传入

配置 ItemWriterAdapter

目前已经有服务（com.juxtapose.example.ch07.reuse.ExistService）可以将信用卡账单信息持久化，接下来我们使用 ExistService 作为示例来演示如何使用 ItemWriterAdapter。

已经存在的服务 ExistService 的示例代码，参见代码清单 7-77。

完整代码参见：com.juxtapose.example.ch07.reuse.ExistService。

代码清单 7-77 已存在服务 ExistService 示例代码

```
1. public class ExistService {
2.     List<CreditBill> billList = new ArrayList<CreditBill>();
3.
4.     public void insert(CreditBill creditBill){
5.         billList.add(creditBill);
6.         System.out.println("ExistService insert:" + creditBill.toString());
7.     }
8.
9.     public void insert(String accountID, String name, double amount,
10.         String date, String address) {
11.         CreditBill creditBill = new CreditBill(accountID, name, amount, date,
            address);
```



```

12.         billList.add(creditBill);
13.         System.out.println("ExistService insert:" + creditBill.toString());
14.     }
15. }

```

本服务模拟已经存在的服务，共有两个不同的 insert 操作，4~7 行使用 CreditBill 作为参数，9~14 行使用多个基本类型作为参数；本例中将介绍如何使用 CreditBill 作为参数的 insert 操作。

接下来我们学习如何配置 ItemWriterAdapter。配置代码参见代码清单 7-78。

完整配置参见文件：ch07/job/job-reuse-service.xml。

代码清单 7-78 配置 ItemWriterAdapter

```

1.     <bean:bean id="reuseServiceWriter"
2.         class="org.springframework.batch.item.adapter.
           ItemWriterAdapter">
3.         <bean:property name="targetObject" ref="existService"/>
4.         <bean:property name="targetMethod" value="insert"/>
5.     </bean:bean>
6.     <bean:bean id="existService"
7.         class="com.juxtapose.example.ch07.reuse.ExistService"/>

```

其中，1~5 行：复用现有的服务完成 ItemWriterAdapter 的配置，属性 targetObject 使用定义的已存服务 existService；属性 targetMethod 指定调用 insert 操作（参数为 CreditBill 的）。

6~7 行：声明现存的服务。

截至目前我们配置完了如何使用现有的服务，通过 ItemWriterAdapter 可以轻松方便地使用现有的服务功能，避免重复发明新的轮子。

使用代码清单 7-79 执行定义的 reuseServiceJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchReuseServiceWrite。

代码清单 7-79 执行 reuseServiceJob

```

1. executeJob("ch07/job/job-reuse-service.xml", "reuseServiceJob",
2.     new JobParametersBuilder().addDate("date", new Date()));

```

7.10.2 PropertyExtractingDelegatingItemWriter

PropertyExtractingDelegatingItemWriter 代理的服务支持更复杂的参数，参数可以根据指定的属性值从 Item 中抽取（ItemWriterAdapter 仅支持参数类型为具体的 Item 对象）。我们仍然使用上节中提供的服务，这次使用多个基本类型作为参数的 insert 操作。

PropertyExtractingDelegatingItemWriter 结构关键属性

PropertyExtractingDelegatingItemWriter 持有服务对象，并调用指定的操作来完成 ItemWriter 中定义的 write 功能。需要注意的是：使用该代理类支持已经存在的服务是复杂类

型的参数，使用接口 BeanWrapper 将给定的 Item 对象进行值的抽取，抽取的值作为存在服务的参数。

PropertyExtractingDelegatingItemWriter 和现有服务之间的关系参见图 7-27。

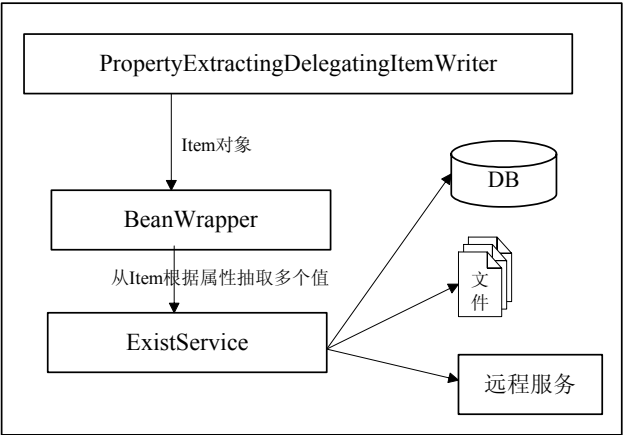


图 7-27 PropertyExtractingDelegatingItemWriter 和现有服务之间的关系

PropertyExtractingDelegatingItemWriter 类关系图参见图 7-28。

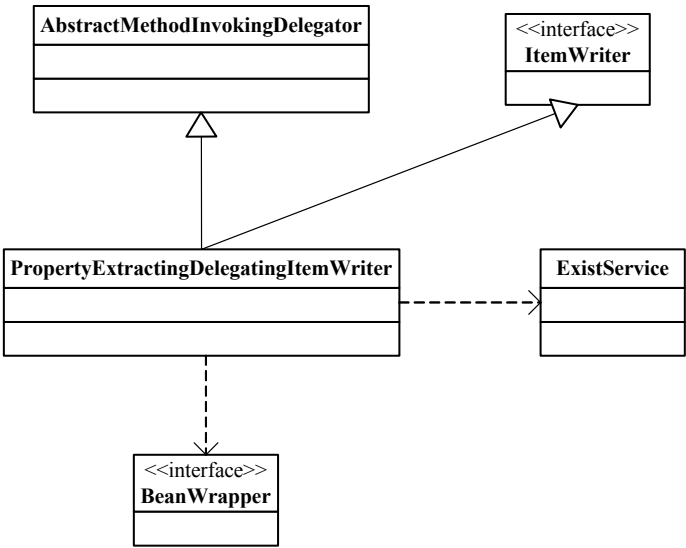


图 7-28 PropertyExtractingDelegatingItemWriter 类关系图

PropertyExtractingDelegatingItemWriter 实现接口 ItemWriter 并继承 AbstractMethodInvokingDelegator，后者提供了调用代理服务的一系列方法；ExistService 表示现存的服务；

BeanWrapper 负责将给定的 Item 对象进行值的抽取，抽取的值作为 ExistService 的参数。PropertyExtractingDelegatingItemWriter 关键属性参见表 7-20。

表 7-20 PropertyExtractingDelegatingItemWriter 关键属性

PropertyExtractingDelegatingItemWriter 属性	类 型	说 明
targetObject	Object	需要调用的目标服务对象
targetMethod	String	需要调用的目标操作名称
fieldsUsedAsTargetMethodArguments	String[]	指定的 Item 中的属性名列表，根据这些指定的属性名从 Item 中抽取属性值，这些属性值被当做调用目标方法的参数值

配置 PropertyExtractingDelegatingItemWriter

接下来我们学习如何配置 PropertyExtractingDelegatingItemWriter。具体配置参见代码清单 7-80。

完整配置参见文件：ch07/job/job-reuse-service.xml。

代码清单 7-80 配置 PropertyExtractingDelegatingItemWriter

```
1.      <bean:bean id="reuseServicePropertyExtractWriter"
2.          class="org.springframework.batch.item.adapter.
3.              PropertyExtractingDelegatingItemWriter">
4.          <bean:property name="targetObject" ref="existService"/>
5.          <bean:property name="targetMethod" value="insert"/>
6.          <bean:property name="fieldsUsedAsTargetMethodArguments">
7.              <bean:list>
8.                  <bean:value>accountID</bean:value>
9.                  <bean:value>name</bean:value>
10.                 <bean:value>amount</bean:value>
11.                 <bean:value>date</bean:value>
12.                 <bean:value>address</bean:value>
13.             </bean:list>
14.          </bean:property>
15.      </bean:bean>
16.      <bean:bean id="existService"
17.          class="com.juxtapose.example.ch07.reuse.ExistService"/>
```

其中，1~15 行：复用现有的服务完成 ItemWriterAdapter 的配置，属性 targetObject 使用定义的已存服务 existService；属性 targetMethod 指定调用 insert 操作（参数为多个基本类型的）。

6 行：定义需要从 Item 中抽取的属性，将抽取出来的属性值对应到目标操作的参数上。

16~17 行：声明现存的服务。

7.11 自定义 ItemWrite

Spring Batch 框架提供了丰富的 ItemWriter 组件, 当这些默认的系统组件不能满足需求时, 我们可以自己实现 ItemWriter 接口来完成需要的业务操作。自定义实现 ItemWriter 非常容易, 只需要实现接口 ItemWriter; 通常只实现接口 ItemWriter 的写不支持重启, 为了支持可重启的自定义 ItemWriter 需要新增实现接口 ItemStream。接下来我们展示如何实现自定义的 ItemWriter。

7.11.1 不可重启 ItemWriter

接口 ItemWriter 的定义, 参见代码清单 7-81。

代码清单 7-81 ItemWriter 接口定义

```
1. public interface ItemWriter<T> {  
2.     void write(List<? extends T> items) throws Exception;  
3. }
```

write 操作批量接受 Item 对象, 一次将所有给定的 Item 持久化到给定的资源中, 每次接受的 items 的大小为在 Chunk 中定义的提交间隔 (commit-interval) 的值。下面的示例展示了自定义 CustomCreditBillItemReader 的实现。

CustomCreditBillItemWriter 的实现参见代码清单 7-82。

完整代码参见: com.juxtapose.example.ch07.cust.itemwriter.CustomCreditBillItemWriter。

代码清单 7-82 CustomCreditBillItemWriter 类定义

```
1. public class CustomCreditBillItemWriter implements ItemWriter<CreditBill> {  
2.     private List<CreditBill> result = TransactionAwareProxyFactory.  
        createTransactionalList();  
3.  
4.     public void write(List<? extends CreditBill> items) throws Exception {  
5.         for(CreditBill item : items){  
6.             result.add(item);  
7.         }  
8.     }  
9.     .....  
10. }
```

其中, 4~7 行: write 操作将传入的 Item 列表持久化到模拟的事务数组中。

配置自定义的 ItemWriter 非常简单, 只需要简单地声明 bean 就可以了。

配置自定义 ItemWriter 的声明, 参见代码清单 7-83。

完整 Job 配置参见文件: ch07/job/job-custom-itemwriter.xml。

代码清单 7-83 配置自定义的 ItemWriter

```
1. <bean:bean id="customItemWriter"  
2.     class="com.juxtapose.example.ch07.cust.itemwriter.
```

```
CustomCreditBillItemWriter">
3.    </bean:bean>
```

其中，1~3 行：声明自定义的 `ItemWriter`。

接下来运行自定义 `ItemWriter`，执行代码参见代码清单 7-84。

完整代码参见类：`test.com.juxtapose.example.ch07.JobLaunchCustomItemWriterTest`。

代码清单 7-84 执行自定义 `ItemWriter`

```
1. List<CreditBill> list = new ArrayList<CreditBill>();
2. list.add(new CreditBill("1", "tom", 100.00, "2013-2-2 12:00:08", "Lu Jia Zui
   road"));
3. list.add(new CreditBill("2", "tom", 320, "2013-2-3 10:35:21", "Lu Jia Zui
   road"));
4. list.add(new CreditBill("3", "tom", 360.00, "2013-2-11 11:12:38", "Longyang
   road"));
5. CustomCreditBillItemWriter writer = new CustomCreditBillItemWriter();
6. writer.write(list);
7. Assert.assertEquals(3, writer.getResult().size());
```

其中，1~4 行：准备示例数据，完成 `list` 的初始化对象。

5 行：完成 `CustomCreditBillItemWriter` 的初始化。

6 行：执行 `write` 操作，将 `Items` 列表持久化。

7 行：断言最终写入的记录条数为 3 条。

本节实现了自定义的 `ItemWriter`，但本节实现的自定义的 `ItemWriter` 不支持重新启动的特性，导致 `Job` 作业失败的情况下不能从失败的执行点重新写入。接下来的章节我们将实现可重启的自定义 `ItemWriter`。

7.11.2 可重启 `ItemWriter`

`Spring Batch` 框架对 `Job` 提供了可重启的能力，`Spring Batch` 框架提供的写组件 `FlatFileItemWriter`、`MultiResourceItemWriter`、`StaxEventItemWriter` 均实现了 `ItemStream` 接口。

和 `ItemReader` 不一样的是，`ItemReader` 的系统读组件基本都实现了 `ItemStream` 接口；而 `ItemWriter` 仅有部分的系统组件实现了 `ItemStream` 接口。因为通常情况下如果写的资源本身是事务性操作的，比如数据库的写，对 `JMS` 消息的写等，如果发生错误之后，因为在事务上下文中会导致整个批量写入都不成功，下次 `Job` 重启的时候，会从失败点继续写入。因此本身具有事务性的写操作不需要实现 `ItemStream` 就支持可重启的特性，例如 `JdbcBatchItemWriter`、`HibernateItemWriter` 等。

如果写操作本身是有状态的，为了支持可重启的特性必须实现 `ItemStream`，例如 `FlatFileItemWriter`、`StaxEventItemWriter`，写入的文件不具有事务的特性。接下来我们学习如何实现可重启的自定义的 `ItemWriter` 实现。

`ItemStream` 接口定义参见代码清单 7-85。

代码清单 7-85 ItemStream 接口定义

```
1. public interface ItemStream {
2.     void open(ExecutionContext executionContext) throws
        ItemStreamException;
3.     void update(ExecutionContext executionContext) throws
        ItemStreamException;
4.     void close() throws ItemStreamException;
5. }
```

其中，2 行：`open()`操作根据参数 `executionContext` 打开需要写入的资源，可以根据持久化在执行上下文 `executionContext` 中的坐标信息重新定位需要写入记录的位置。

3 行：`update()`操作将需要持久化的数据存放在执行上下文 `executionContext` 中，通常将当前的状态数据保存在上下文中。

4 行：`close()`操作关闭写入的资源。

`ItemStream` 接口定义了写操作与执行上下文 `ExecutionContext` 交互的能力。可以将已经写的条数通过该接口存放在执行上下文 `ExecutionContext` 中（`ExecutionContext` 中的数据在批处理 `commit` 的时候会通过 `JobRepository` 持久化到数据库中），这样到 `Job` 发生异常重新启动 `Job` 的时候，写操作可以跳过已经成功写的的数据，继续从上次出错的地点（可以从执行上下文中获取上次成功写的位置）开始写。

接下来我们改造上节的自定义的 `ItemWriter`，新增实现接口 `ItemStream`，使其支持可重启的能力。`RestartableCustomCreditBillItemWriter` 的实现参见代码清单 7-86。

完整代码参见：`com.juxtapose.example.ch07.cust.itemwriter.RestartableCustomCreditBillItemWriter`。

代码清单 7-86 RestartableCustomCreditBillItemWriter 类定义

```
1. public class RestartableCustomCreditBillItemWriter implements
2.     ItemWriter<CreditBill>, ItemStream {
3.     private List<CreditBill> result = new ArrayList<CreditBill>();
4.     private int currentLocation = 0;
5.     private static final String CURRENT_LOCATION = "current.location";
6.
7.     public void write(List<? extends CreditBill> items) throws Exception {
8.         for(;currentLocation < items.size();){
9.             result.add(items.get(currentLocation++));
10.        }
11.    }
12.
13.    public void open(ExecutionContext executionContext)
14.        throws ItemStreamException {
15.        if(executionContext.containsKey(CURRENT_LOCATION)){
16.            currentLocation = new Long(executionContext.
17.                getLong(CURRENT_LOCATION)).intValue();
```

```

18.         }else{
19.             currentLocation = 0;
20.         }
21.     }
22.
23.     public void update(ExecutionContext executionContext)
24.         throws ItemStreamException {
25.         executionContext.putLong(CURRENT_LOCATION,
26.                                 new Long(currentLocation).longValue());
27.     }
28.
29.     public void close() throws ItemStreamException {}
30. }

```

其中，7~11 行：**write** 操作，根据当前的位置 **currentLocation** 从 **list** 中读取数据并写入持久化资源中，当前的位置标识在执行上下文中获取，具体参见 **open** 操作中的代码实现。

13~21 行：**open** 操作，从执行上下文中获取当前写入的位置。

23~27 行：**update** 操作，将当前已经写过的数据位置存放在执行上下文中，通常 **update** 操作在 **chunk** 的事务提交后会执行一次。

29 行：**close** 操作，通常在此处关闭不再需要的资源。

配置自定义的可重启 **ItemWriter** 非常简单，只需要简单的声明 **bean** 就可以。

配置自定义 **ItemWriter** 的声明，参见代码清单 7-87。

完整 **Job** 配置参见文件：**ch07/job/job-custom-itemwriter.xml**。

代码清单 7-87 配置自定义 **ItemWriter**

```

1. <bean:bean id="restartableCustomItemWriter"
2.           class="com.juxtapose.example.ch07.cust.itemwriter.
3.               RestartableCustomCreditBillItemWriter">
4. </bean:bean>

```

其中，1~4 行：声明自定义的可重启的 **ItemWriter**。

接下来运行 **RestartableCustomCreditBillItemWriter**，执行代码参见代码清单 7-88。

完整代码参见类：**test.com.juxtapose.example.ch07.JobLaunchCustomItemWriterTest**。

代码清单 7-88 执行自定义 **ItemWriter**

```

1. List<CreditBill> list = new ArrayList<CreditBill>();
2. list.add(new CreditBill("1", "tom", 100.00, "2013-2-2 12:00:08", "Lu Jia Zui
   road"));
3. RestartableCustomCreditBillItemWriter writer = new RestartableCustom
   CreditBillItemWriter();
4. ExecutionContext executionContext = new ExecutionContext();
5. ((ItemStream)writer).open(executionContext);
6. writer.write(list);
7. Assert.assertEquals(1, writer.getResult().size());

```

```

8. ((ItemStream)writer).update(executionContext);
9.
10. list.add(new CreditBill("2","tom",320,"2013-2-3 10:35:21","Lu Jia Zui
    road"));
11. list.add(new CreditBill("3","tom",360.00,"2013-2-11 11:12:38","Longyang
    road"));
12. writer = new RestartableCustomCreditBillItemWriter();
13. ((ItemStream)writer).open(executionContext);
14. writer.write(list);
15. Assert.assertEquals(2, writer.getResult().size());
16. ((ItemStream)writer).update(executionContext);

```

其中，1~2 行：准备示例数据，完成 list 的初始化对象。

3 行：完成 RestartableCustomCreditBillItemWriter 的初始化。

4 行：模拟新建执行上下文对象 executionContext。

5 行：执行自定义 ItemWriter 的 open 操作，从执行上下文中获取当前已写入记录的位置。

6 行：执行写操作。

7 行：断言写入后只有一条记录。

8 行：执行自定义 ItemWriter 的 update 操作，将当前记录的位置存放在执行上下文中。

10~11 行：为 list 增加两条记录。

12 行：重新构造一个新的 RestartableCustomCreditBillItemWriter 对象，模拟重启该 Writer。

13 行：使用同一个执行上下文执行 open 操作。

14 行：再次执行写操作，这次会从上次执行的位置来写数据，因此本次写入的总记录数应该为 2 条。

15 行：断言本次写入的行数为 2 行。

本示例代码模拟了重新启动写操作的场景，其本质是运行时将游标的位置存放在执行上下文中，执行上下文中的数据在每次事务提交的时候会保存到数据库中；当作业 Job 重新启动的时候从执行上下文中重新获取上次写操作的位置，从正确的位置开始写操作，从而完成了支持重启的能力。

7.12 拦截器

Spring Batch 框架在 ItemWriter 执行阶段提供了拦截器，使得在 ItemWriter 执行前后能够加入自定义的业务逻辑。ItemWriter 执行阶段拦截器接口为：org.springframework.batch.core.ItemWriteListener<S>。

7.12.1 拦截器接口

接口 ItemWriteListener 的定义参见代码清单 7-89。

代码清单 7-89 ItemWriteListener 接口定义

```
1. public interface ItemWriteListener<S> extends StepListener {
2.     void beforeWrite(List<? extends S> items);
3.     void afterWrite(List<? extends S> items);
4.     void onError(Exception exception, List<? extends S> items);
5. }
```

其中，2 行：beforeWrite 在写入资源前触发该操作。

3 行：afterWrite 在写入资源后触发该操作。

4 行：onWriteError 在写入资源发生异常的时候触发该操作。

为 ItemWriter 配置拦截器，具体配置参见代码清单 7-90。

完整配置参见文件：/ch07/job/job-listener.xml。

代码清单 7-90 配置 ItemWriter 拦截器

```
1.     <job id="itemReadJob">
2.         <step id="itemReadStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="creditItemRead" processor="creditBillProcessor"
5.                     writer="creditItemWriter" commit-interval="2">
6.                     <listeners>
7.                         <listener ref="sysoutItemWriteListener"></listener>
8.                         <listener ref="sysoutAnnotationItemWriteListener">
9.                             </listeners>
10.                    </chunk>
11.                    <listeners>
12.                    </listeners>
13.                </tasklet>
14.            </step>
15.        </job>
16.        <bean:bean id="sysoutItemWriteListener"
17.            class="com.juxtapose.example.ch07.listener.
18.                SystemOutItemWriteListener">
19.        </bean:bean>
20.        <bean:bean id="sysoutAnnotationItemWriteListener"
21.            class="com.juxtapose.example.ch07.listener.SystemOutAnnotation">
22.        </bean:bean>
```

其中，6~9 行：为作业的读配置 2 个拦截器。

16~18 行：定义拦截器 sysoutItemWriteListener，该拦截器实现接口 ItemWriteListener。

20~22 行：定义拦截器 sysoutAnnotationItemWriteListener，该拦截器通过 Annotation 方式定义。

SystemOutItemWriteListener 的代码参见代码清单 7-91。

类 `SystemOutItemWriteListener` 完整代码参见：`com.juxtapose.example.ch07.listener.SystemOutItemWriteListener`。

代码清单 7-91 `SystemOutItemWriteListener` 类定义

```
1. public class SystemOutItemWriteListener implements
   ItemWriteListener <CreditBill> {
2.     public void beforeWrite(List<? extends CreditBill> items) {
3.         System.out.println("SystemOutItemWriteListener.beforeWrite()");
4.     }
5.
6.     public void afterWrite(List<? extends CreditBill> items) {
7.         System.out.println("SystemOutItemWriteListener.afterWrite()");
8.     }
9.
10.    public void onWriteError(Exception exception,
11.        List<? extends CreditBill> items) {
12.        System.out.println("SystemOutItemWriteListener.onWriteError()");
13.    }
14. }
```

7.12.2 拦截器异常

拦截器方法如果抛出异常会影响 `Job` 的执行，所以在执行自定义的拦截器时候，需要考虑对拦截器发生的异常做处理，避免影响业务。

如果拦截器发生异常，会导致 `Job` 执行的状态为"FAILED"。

配置了错误拦截器的作业配置参见代码清单 7-92。

完整配置参见文件：`/ch07/job/job-listener.xml`。

代码清单 7-92 配置了错误拦截器的作业

```
1. <job id="errorItemReadJob">
2.     <step id="errorItemReadStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="creditItemRead" processor="creditBillProcessor"
5.                 writer="creditItemWriter" commit-interval="2">
6.                 <listeners>
7.                     <listener ref="errorItemWriteListener"></listener>
8.                 </listeners>
9.             </chunk>
10.         </tasklet>
11.     </step>
12. </job>
```

其中，7 行：`errorItemWriteListener` 在 `beforeWrite` 操作中抛出异常，会导致整个作业的

失败，当前的 Job 实例状态被标记为" FAILED "。

7.12.3 执行顺序

在配置文件中可以配置多个 ItemWriteListener，拦截器之间的执行顺序按照 listeners 定义的顺序执行。beforeWrite 方法按照 listener 定义的顺序执行，afterWrite 方法按照相反的顺序执行。上面示例代码中执行顺序如下：

- (1) sysoutItemWriteListener 拦截器的 beforeWrite 方法；
- (2) sysoutAnnotationItemWriteListener 拦截器的 beforeWrite 方法；
- (3) sysoutAnnotationItemWriteListener 拦截器的 afterWrite 方法；
- (4) sysoutItemWriteListener 拦截器的 afterWrite 方法。

7.12.4 Annotation

Spring Batch 框架提供了 Annotation 机制，可以不实现接口 ItemWriteListener，直接通过 Annotation 的机制定义拦截器。为 ItemWriteListener 提供的 Annotation 有：

- @BeforeWrite；
- @AfterWrite；
- @OnWriteError。

ItemWriteListener 操作说明与 Annotation 定义参见表 7-21。

表 7-21 ItemWriteListener 操作说明与 Annotation 定义

操 作	操作说明	Annotation
beforeWrite(List<? extends S> items)	在 ItemWriter#write()之前执行	@ BeforeWrite
afterWrite(List<? extends S> items)	在 ItemWriter# write ()之后执行	@ AfterWrite
onWriteError(Exception exception, List<? extends S> items)	当 ItemWriter# write ()抛出异常时候触发该操作	@ OnWriteError

使用 Annotation 声明的拦截器的 Spring 配置文件和实现接口 ItemWriteListener 的拦截器配置一样，只需要在 listeners 节点中声明即可。

SystemOutAnnotation 的代码参见代码清单 7-93。

类 SystemOutAnnotation 完整代码参见：com.juxtapose.example.ch07.listener.SystemOut Annotation。

代码清单 7-93 SystemOutAnnotation 类定义

```
1. public class SystemOutAnnotation {
2.     @BeforeWrite
3.     public void beforeWrite(List<? extends CreditBill> items) {
4.         System.out.println("SystemOutAnnotation.beforeWrite()");
5.     }
6. }
```

```

5.     }
6.
7.     @AfterWrite
8.     public void afterWrite(List<? extends CreditBill> items) {
9.         System.out.println("SystemOutAnnotation.afterWrite()");
10.    }
11.
12.    @OnWriteError
13.    public void onWriteError(Exception exception,
14.        List<? extends CreditBill> items) {
15.        System.out.println("SystemOutAnnotation.onWriteError()");
16.    }
17. }

```

7.12.5 属性 Merge

Spring Batch 框架提供了多处配置拦截器执行，可以在 `chunk` 元素节点配置，也可以在 `tasklet` 中配置；框架同样提供了 `step` 的抽象和继承的能力，可以在父 `Step` 中定义通用的属性，在子 `step` 中定义个性化的属性，通过 `merge` 属性可以定义是覆盖父中的设置、还是和父中的定义合并；`chunk` 元素中的 `listeners` 支持 `merge` 属性。

假设有这样一个场景，所有的 `Step` 都希望拦截器 `sysoutItemWriteListener` 能够执行，而拦截器 `sysoutAnnotationItemWriteListener` 则由每个具体的 `Step` 定义是否执行，通过抽象和继承属性可以完成上面的场景。

`merge` 属性配置代码参见代码清单 7-94。

代码清单 7-94 `merge` 属性配置

```

1. <job id="mergeChunkJob">
2.     <step id="subChunkStep" parent="abstractParentStep">
3.         <tasklet>
4.             <chunk reader="creditItemRead" processor="creditBillProcessor"
5.                 writer="creditItemWriter" >
6.                 <listeners merge="true">
7.                     <listener ref="sysoutAnnotationItemWriteListener">
8.                         </listener>
9.                     </listeners>
10.                </chunk>
11.            </tasklet>
12.        </step>
13.    </job>
14.    <step id="abstractParentStep" abstract="true">
15.        <tasklet>

```

```
16.         <chunk commit-interval="2" >
17.             <listeners>
18.                 <listener ref="sysoutItemWriteListener"></listener>
19.             </listeners>
20.         </chunk>
21.     </tasklet>
22. </step>
```

其中，17~18 行：定义抽象作业步 `abstractParentStep` 中的拦截器。

6~8 行：通过 `merge` 属性，可以与父类中的拦截器配置进行合并，表示在 `subChunkStep` 中有两个拦截器会同时工作；属性 `merge` 的值为 `true`，表示父子合并即两处定义的都生效；实行 `merge` 的值为 `false`，表示父类的不再生效，被子类的定义覆盖掉了。

通过 `merge` 属性合并的拦截器的执行顺序如下：首先执行父 `Step` 中定义的拦截器；然后执行子 `Step` 中定义的拦截器。

处理数据 ItemProcessor

批处理通过 Tasklet 完成具体的任务，chunk 类型的 tasklet 定义了标准的读、处理、写的执行步骤。批处理在读取数据后，写入数据之前，希望能够提供一个处理数据的阶段，ItemProcessor 是实现处理阶段的重要组件，Spring Batch 框架提供了丰富的处理组件，包括数据转换、组合处理、数据过滤、数据校验等能力，和前面的数据读取和写入一样，在处理数据阶段框架同样提供了复用现有业务逻辑的能力。

8.1 ItemProcessor

ItemProcessor 是 Step 中对资源的处理阶段，Spring Batch 框架已经提供了各种类型的处理实现，包括包括数据转换、组合处理、数据过滤、数据校验等。

处理操作与 Chunk Tasklet 的关系图参见图 8-1。

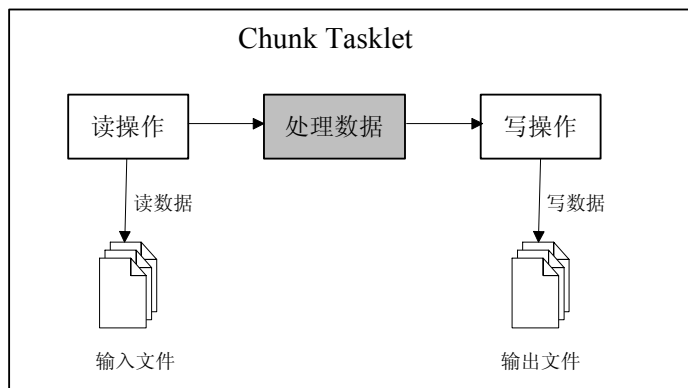


图 8-1 处理操作与 Chunk Tasklet 的关系图

需要注意的是数据处理阶段是可选的，也就是说可以只有读、写操作而没有中间的处理数据的阶段；这种情况下读的数据会直接交给写阶段来处理。

8.1.1 ItemProcessor

所有的处理操作都需要实现 `org.springframework.batch.item.ItemProcessor<I, O>` 接口。ItemProcessor 接口定义参见代码清单 8-1。

代码清单 8-1 ItemProcessor 接口定义

```
1. public interface ItemProcessor<I, O> {  
2.     O process(I item) throws Exception;  
3. }
```

其中，2 行：ItemProcessor 接口定义了核心作业方法 process () 操作，参数 I item 是读阶段获取的对象；返回值 O 会提供给写阶段，作为写阶段的输入参数。

读者可能会注意到 ItemWrite 接口定义的写操作的参数是 List<? extends T> items，Spring Batch 框架会将 ItemProcessor 阶段处理的数据收集起来，直到满足在 chunk 中定义的提交间隔（commit-interval）的大小才将处理的数据批量提交给 ItemWrite 进行处理。

读、处理、写操作三者间的数据转换关系图参见图 8-2。

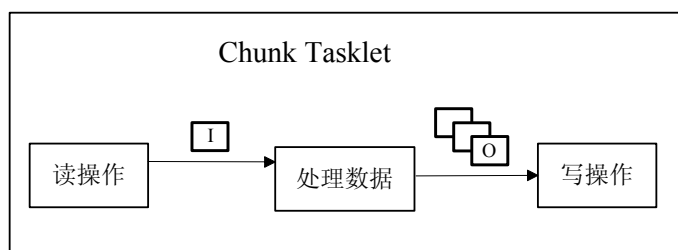


图 8-2 读、处理、写操作三者间的数据转换关系图

负责将 ItemReader 读入的数据进行处理后写入指定的资源中；注意这里 write 操作的参数是 List<? extends T> items 类型的，表示写操作通常会进行批量的写入；每次写入 List 的大小由属性 commit-interval 决定。

Job 中典型的配置 ItemProcessor 参见代码清单 8-2。

代码清单 8-2 典型的 ItemProcessor 配置

```
1. <job id="dbReadJob">  
2.     <step id="dbReadStep">  
3.         <tasklet transaction-manager="transactionManager">  
4.             <chunk reader="jdbcParameterItemReader" processor=  
5.                 "creditBillProcessor"  
6.                 writer="creditItemWriter" commit-interval="2"></chunk>  
7.         </tasklet>  
8.     </step>  
9. </job>
```

8.1.2 系统处理组件

Spring Batch 框架提供的处理组件列表参见表 8-1。

表 8-1 Spring Batch 框架提供的处理组件

ItemProcessor	说 明
CompositeItemProcessor	组合处理器，可以封装多个业务处理服务
ItemProcessorAdapter	ItemProcessor 适配器，可以复用现有的业务处理服务
PassThroughItemProcessor	不做任何业务处理，直接返回读到的数据
ValidatingItemProcessor	数据校验处理器，支持对数据的校验，如果校验不通过可以进行过滤掉或者通过 skip 的方式跳过对记录的处理

8.2 数据转换

ItemProcessor 的一个核心作用是对读阶段的数据进行转换，其中包括对部分数据进行更改，还可以根据读的数据完全返回一个不同类型的数据给处理阶段。

8.2.1 部分数据转换

部分数据转换效果图参见图 8-3。

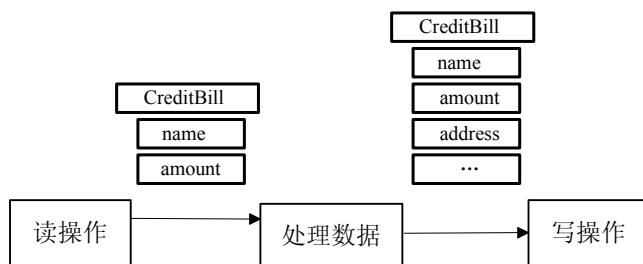


图 8-3 部分数据转换效果图

在部分转换的情况下，不会更改读入阶段的数据类型，可以针对读出的数据进行属性值的重新修订或者重新计算；在数据库的典型操作中甚至可以根据主键到数据库中进行查询，重新生成一个相同类型的数据对象。

需要处理的记录参见代码清单 8-3。

代码清单 8-3 需要处理的数据清单

```

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road

```

接下来我们展示如何使用 ItemProcessor 进行数据转换操作，PartTranslateItemProcessor

实现 `ItemProcessor` 接口，负责将读入的对象 `CreditBill` 进行部分属性的变更。`PartTranslateItemProcessor` 的实现代码参见代码清单 8-4。

完整代码参见：`com.juxtapose.example.ch08.PartTranslateItemProcessor`。

代码清单 8-4 `PartTranslateItemProcessor` 类定义

```
1. public class PartTranslateItemProcessor implements
2.     ItemProcessor<CreditBill, CreditBill> {
3.
4.     public CreditBill process(CreditBill bill) throws Exception {
5.         bill.setAddress(bill.getAddress() + "," + bill.getName());
6.         .....
7.         return bill;
8.     }
9. }
```

配置文件声明参见代码清单 8-5。

完整配置参见：`ch08/job/job-translate.xml`。

代码清单 8-5 配置 `partTranslateJob`

```
1.     <job id="partTranslateJob">
2.         <step id="partTranslateStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="flatFileItemReader"
5.                     writer="part TranslateFlatFileItemWriter"
6.                     processor="partTranslateItemProcessor"
7.                     commit-interval="2">
8.                 </chunk>
9.             </tasklet>
10.         </step>
11.     </job>
12.     <bean:bean id="partTranslateItemProcessor"
13.         class="com.juxtapose.example.ch08.PartTranslateItemProcessor">
14.     </bean:bean>
```

其中，10~12 行：声明数据转换的定义。

经过部分数据处理后写入的文件内容参见代码清单 8-6。

代码清单 8-6 转换后的数据清单

```
1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road,tom
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road,tom
3. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road,tom
4. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road,tom
5. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road,tom
6. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road,tom
```

每行的记录最后多了",tom"。

8.2.2 数据类型转换

数据类型转换效果图参见图 8-4。

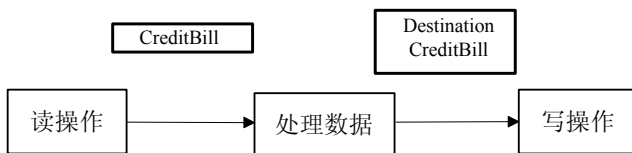


图 8-4 数据类型转换效果图

在数据类型变换的情况下，通常是读、写阶段需要处理的数据类型不一致，此时可以通过 `ItemProcessor` 进行数据的适配，例如图 8-4 中读阶段获取的对象为 `CreditBill` 类型，通过数据处理后返回给写操作的类型为 `DestinationCreditBill` 的对象。

接下来我们展示如何使用 `ItemProcessor` 进行数据转换操作，`TranslateItemProcessor` 实现 `ItemProcessor` 接口，负责将读入的对象 `CreditBill` 转换为写阶段需要处理的对象 `DestinationCreditBill`。`TranslateItemProcessor` 类实现参见代码清单 8-7。

完整代码参见：`com.juxtapose.example.ch08.TranslateItemProcessor`。

代码清单 8-7 `TranslateItemProcessor` 类定义

```
1. public class TranslateItemProcessor implements
2.     ItemProcessor<CreditBill, DestinationCreditBill> {
3.
4.     public DestinationCreditBill process(CreditBill bill) throws Exception {
5.         DestinationCreditBill destCreditBill = new DestinationCreditBill();
6.         destCreditBill.setAccountID(bill.getAccountID());
7.         destCreditBill.setAddress(bill.getAddress());
8.         .....
9.         return destCreditBill;
10.    }
11. }
```

配置文件声明参见代码清单 8-8。

完整配置参见：`ch08/job/job-translate.xml`。

代码清单 8-8 配置 `translateJob`

```
1. <job id="translateJob">
2.     <step id="translateStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="flatFileItemReader"
5.                 writer="translateFlatFileItemWriter"
6.                 processor="translateItemProcessor"
7.                 commit-interval="2">
```

```

7.         </tasklet>
8.     </step>
9. </job>
10. <bean:bean id="translateItemProcessor"
11.     class="com.juxtapose.example.ch08.TranslateItemProcessor">
12. </bean:bean>

```

其中，10~12 行：声明数据转换的定义。

8.3 数据过滤

数据处理除了支持上节的数据转换功能外，同样对数据提供了过滤的能力。过滤是指如果对读入阶段的数据不期望在写入阶段被写入，可以通过返回 `null` 来阻止该 `Item` 数据被写入。

8.3.1 数据 Filter

数据过滤的功能参见图 8-5。



图 8-5 数据过滤示意图

仍然使用处理信用卡账单的例子，所有金额大于 500 的数据需要过滤掉，不能被提交到 `write` 阶段写入到文件中。接下来我们实现数据过滤的 `FilterItemProcessor`。

需要处理的记录参见代码清单 8-9。

代码清单 8-9 需要处理的数据清单

```

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road

```

`FilterItemProcessor` 的实现参见代码清单 8-10。

完整代码参见：`com.juxtapose.example.ch08.FilterItemProcessor`。

代码清单 8-10 `FilterItemProcessor` 类定义

```

1. public class FilterItemProcessor implements ItemProcessor<CreditBill,
   CreditBill> {
2.

```

```

3.     @Override
4.     public CreditBill process(CreditBill item) throws Exception {
5.         if(item.getAmount() > 500){
6.             return null;
7.         }else{
8.             return item;
9.         }
10.    }
11. }

```

配置文件声明参见代码清单 8-11。

完整配置参见：ch08/job/job-filter.xml。

代码清单 8-11 配置 filterJob

```

1.     <job id="filterJob">
2.         <step id="filterStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="flatFileItemReader" writer="flatFileItemWriter"
5.                     processor="filterItemProcessor" commit-interval="2">
6.                     </chunk>
7.                 </tasklet>
8.             </step>
9.         </job>
10.    <bean:bean id="filterItemProcessor"
11.        class="com.juxtapose.example.ch08.FilterItemProcessor">
12.    </bean:bean>

```

其中，1~9 行：定义 Job，使用数据过滤功能。

10~12 行：声明数据过滤的定义。

经过数据过滤后写入的文件内容参见代码清单 8-12。

代码清单 8-12 过滤后的数据清单

```

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road

```

数据过滤与异常跳过的区别

(1) 数据过滤是指 ItemProcessor 中的处理操作如果返回值是 Null，则该条数据被过滤掉，不会进入到 ItemWrite 阶段。

(2) 异常跳过是指在 Item 的处理阶段如果发生了指定类型的异常，则该条记录被忽略掉。

8.3.2 数据过滤统计

无论是数据过滤还是异常跳过处理，在 Job 执行期间都会把源信息存入到 JobRepository

中；可以通过作业步执行器获取被过滤的记录总数、异常跳过的记录总数等信息。

`StepExecution.getFilterCount()` 可以获取被过滤的记录总数。

`StepExecution.getSkipCount()` 可以获取异常跳过的记录总数。

接下来我们展示通过拦截器 `FilterCountStepExecutionListener`（实现作业步拦截器接口）获取被过滤的记录条数。

`FilterCountStepExecutionListener` 的实现参见代码清单 8-13。

完整代码参见：`com.juxtapose.example.ch08.FilterCountStepExecutionListener`。

代码清单 8-13 `FilterCountStepExecutionListener` 类定义

```
1. public class FilterCountStepExecutionListener extends
   StepExecutionListenerSupport {
2.     @Override
3.     public ExitStatus afterStep(StepExecution stepExecution) {
4.         int filterCount = stepExecution.getFilterCount();
5.         System.out.println("Filter count=" + filterCount);
6.         return stepExecution.getExitStatus();
7.     }
8. }
```

其中，4 行：通过作业步执行器 `stepExecution` 获取过滤的记录总数。

配置文件声明参见代码清单 8-14。

完整配置参见：`ch08/job/job-filter.xml`。

代码清单 8-14 配置 `filterJob`

```
1. <job id="filterJob">
2.     <step id="filterStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="flatFileItemReader" writer="flatFileItemWriter"
5.                 processor="filterItemProcessor" commit-interval="2">
6.                 <listeners>
7.                     <listener ref="filterCountStepExecutionListener" />
8.                 </listeners>
9.             </chunk>
10.        </tasklet>
11.    </step>
12. </job>
13. <bean:bean id="filterCountStepExecutionListener"
14.     class="com.juxtapose.example.ch08.
15.         FilterCountStepExecutionListener">
16. </bean:bean>
```

其中，1~12 行：定义 Job，使用数据过滤功能，同时增加了作业步执行器的拦截器。

13~15 行：声明数据过滤的拦截器，负责收集被过滤数据的总条数。

执行该 Job 后，统计到被过滤的数据信息参见代码清单 8-15。

```
1. Filter count=3
```

该 Job 中所有被过滤的记录条目为 3 条。

8.4 数据校验

在业务数据处理过程中经常需要对输入的数据进行有效性校验，例如对于信用卡交易记录显然账号不能为空，需要还款的交易金额不能小于 0 等业务规则。Spring 框架提供了对数据校验的接口，如果校验不通过可以抛出给定类型的异常。

Spring Batch 框架提供了数据校验处理类 `ValidatingItemProcessor`，可以在处理的阶段进行数据校验，`ValidatingItemProcessor` 本身支持过滤的功能和跳过两种能力。

8.4.1 Validator

Spring 框架提供的校验接口为 `org.springframework.batch.item.validator.Validator<T>`，仅有一个操作 `validate`，对输入的参数进行数据校验，如果校验不通过可以抛出类型为 `ValidationException` 的异常。

接口 `Validator` 定义参见代码清单 8-16。

代码清单 8-16 `Validator` 接口定义

```
1. public interface Validator<T> {
2.     void validate(T value) throws ValidationException;
3. }
```

对于信用卡交易，在统计账单的时候仅需要统计消费的部分，而对于客户直接存入的交易部分则不需要统计（通常用负数表示）。因此针对信用卡消费单的数据作如下的校验，如果消费金额小于 0 则不进行数据的处理。

`CreditBillValidator` 的实现参见代码清单 8-17。

完整代码参见：`com.juxtapose.example.ch08.CreditBillValidator`。

代码清单 8-17 `CreditBillValidator` 类定义

```
1. public class CreditBillValidator implements Validator<CreditBill> {
2.
3.     @Override
4.     public void validate(CreditBill creditBill) throws ValidationException {
5.         if(Double.compare(0, creditBill.getAmount()) >0) {
6.             throw new ValidationException("Credit bill cannot be negative!");
7.         }
8.     }
9. }
```

如果消费金额小于 0，则抛出类型为 `ValidationException` 的异常。

代码清单 8-18 是我们本节示例用到的数据记录。

代码清单 8-18 待校验的数据记录清单

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,-674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,-793.20,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,-893.00,2013-2-28 20:34:19,Hunan road
```

负数的记录行需要通过校验的方式跳过处理。

8.4.2 ValidatingItemProcessor

ValidatingItemProcessor 结构关键属性

ValidatingItemProcessor 实现接口 ItemProcessor，通过引用接口 Validator 进行数据校验功能，根据业务需要自定义实现符合业务需求的校验器。ValidatingItemProcessor 支持过滤的功能和跳过两种能力，通过属性 filter 进行标识，true 表示校验不通过的时候直接返回 null，跳出该条记录的处理；false 表示使用异常跳过的能力，可以通过配置 skippable-exception-classes 的方式忽略校验异常。ValidatingItemProcessor 核心类结构图参见图 8-6。

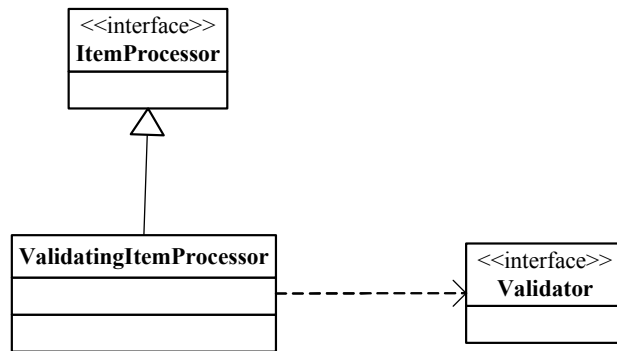


图 8-6 ValidatingItemProcessor 核心类结构图

ValidatingItemProcessor 关键属性参见表 8-2。

表 8-2 ValidatingItemProcessor 关键属性

ValidatingItemProcessor 属性	类 型	说 明
validator	Validator<? super T>	数据校验器，校验给定的 Item 数据是否合法
Filter	Boolean	是否使用过滤功能。 默认值：false

配置 ValidatingItemProcessor

首先定义 ValidatingItemProcessor，参见代码清单 8-19 中的代码。

完整配置文件参见：ch08/job/job-validate.xml。

代码清单 8-19 配置 ValidatingItemProcessor

```
1.     <bean:bean id="validatorProcessor" scope="step"
2.         class="org.springframework.batch.item.validator.
3.             ValidatingItemProcessor">
4.         <bean:property name="filter" value="#{jobParameters['filter']}" />
5.         <bean:property name="validator">
6.             <bean:bean class="com.juxtapose.example.ch08.CreditBillValidator" />
7.         </bean:property>
8.     </bean:bean>
```

其中，3 行：属性 filter 用于声明是否采用过滤的功能，value 值采用了参数后绑定的技术，在执行 Job 的时候需要指定参数"filter"的值。

4~6 行：属性 validator 定义参数校验的具体实现类，使用上节定义的 com.juxtapose.example.ch08.CreditBillValidator。

Job 的配置参见代码清单 8-20。

完整配置文件参见：ch08/job/job-validate.xml。

代码清单 8-20 配置 validateJob

```
1.     <job id="validateJob">
2.         <step id="validateStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="flatFileItemReader" writer="flatFileItemWriter"
5.                     processor="validatorProcessor" commit-interval="2"
6.                     skip-limit="5">
7.                     <skippable-exception-classes>
8.                         <include class="org.springframework.batch.
9.                             item.validator.ValidationException" />
10.                    </skippable-exception-classes>
11.                    <listeners>
12.                        <listener ref="filterCountStepExecutionListener" />
13.                        <listener ref="skipCountStepExecutionListener" />
14.                    </listeners>
15.                </chunk>
16.            </tasklet>
17.        </step>
18.    </job>
```

其中，6~9 行：skippable-exception-classes 定义处理阶段发生异常的会被忽略掉，ValidationException 类型的异常不会引起 Job 作业失败。

11 行：声明过滤统计作业步执行器，实现参见 8.3.2 章节。

12 行：声明跳过统计作业步执行器，skipCountStepExecutionListener 实现参见下面的

SkipCountStepExecutionListener。

SkipCountStepExecutionListener 的实现参见代码清单 8-21。

完整代码参见：com.juxtapose.example.ch08.SkipCountStepExecutionListener。

代码清单 8-21 SkipCountStepExecutionListener 类定义

```
1. public class SkipCountStepExecutionListener extends StepExecutionListener
   Support {
2.     @Override
3.     public ExitStatus afterStep(StepExecution stepExecution) {
4.         int skipCount = stepExecution.getSkipCount();
5.         System.out.println("Skip count=" + skipCount);
6.         return stepExecution.getExitStatus();
7.     }
8. }
```

其中，4 行：通过作业步执行器 stepExecution 获取跳过的记录总数。

执行 Job，经过数据校验后的文件内容参见代码清单 8-22。

代码清单 8-22 数据校验后的文件内容清单

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
```

8.5 组合处理器

在 Spring Batch 框架中对于 Chunk 只能配置一个 ItemProcessor，但在有些业务场景中需要将一个 Item 同时执行多个不同处理器，例如首先进行 Item 的转换，然后再进行数据的校验工作；Spring Batch 框架提供了组合 ItemProcessor（CompositeItemProcessor）的模式满足上面的需求。组合 ItemProcessor 完成的功能参见图 8-7。

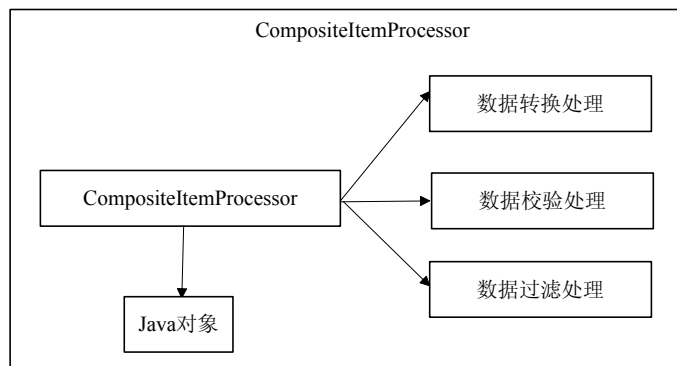


图 8-7 组合 ItemProcessor 完成的功能示意图

组合处理器代理不同的数据处理类，按照定义的顺序先后执行处理器，完成多个处理器的功能。

CompositeItemProcessor 结构关键属性

CompositeItemProcessor 组件实现 ItemProcessor 接口，并引用一组 ItemProcessor 实现类，CompositeItemProcessor 在进行处理阶段循环调用 ItemProcessor 中的实现，实现执行多个处理器的能力。CompositeItemProcessor 核心类结构图参见图 8-8。

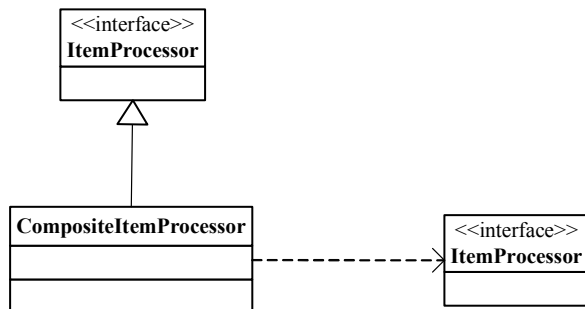


图 8-8 CompositeItemProcessor 核心类结构图

CompositeItemProcessor 关键属性参见表 8-3。

表 8-3 CompositeItemProcessor 关键属性

CompositeItemProcessor 属性	类 型	说 明
delegates	List<? extends ItemProcessor<?, ?>>	被代理的 ItemProcessor 的组合，对于每一个 Item 对象会经过每个处理器进行加工

配置 CompositeItemProcessor

接下来我们使用前面章节使用的部分数据转换和数据校验的功能，首先使用数据转换的功能，修改每条记录的地址信息；然后使用数据校验功能需要保证每条记录的消费金额大于 0。通过 CompositeItemProcessor 来实现上面的处理组合。

代码清单 8-23 是我们本节示例用到的数据记录。

代码清单 8-23 需要处理的数据清单

```

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,-674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,-793.20,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,-893.00,2013-2-28 20:34:19,Hunan road
  
```

配置 CompositeItemProcessor，配置代码参见代码清单 8-24。

完整配置文件参见：ch08/job/job-composite.xml。

代码清单 8-24 配置 CompositeItemProcessor

```
1. <bean:bean id="compositeItemProcessor"
2.     class="org.springframework.batch.item.support.
        CompositeItemProcessor">
3.     <bean:property name="delegates">
4.         <bean:list>
5.             <bean:ref bean="partTranslateItemProcessor" />
6.             <bean:ref bean="validatorProcessor" />
7.         </bean:list>
8.     </bean:property>
9. </bean:bean>
```

其中，3 行：属性 delegates 定义需要组合的处理器，本示例使用数据转换处理器与数据校验处理器，两个处理器的具体配置可以参见前面的章节。

5 行：引用部分数据转换处理器，修改 Item 的地址信息。

6 行：引用数据校验处理器，保证 Item 的消费金额要大于 0。

配置 CompositeItemProcessor 的 Job，参见代码清单 8-25。

完整配置文件参见：ch08/job/job-composite.xml。

代码清单 8-25 配置 compositeJob

```
1. <job id="compositeJob">
2.     <step id="compositeStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="flatFileItemReader" writer="flatFileItemWriter"
5.                 processor="compositeItemProcessor" commit-interval="2"
6.                 skip-limit="5">
7.                 <skippable-exception-classes>
8.                     <include class="org.springframework.batch.item.
9.                         validator.ValidationException"/>
10.                </skippable-exception-classes>
11.                <listeners>
12.                    <listener ref="filterCountStepExecutionListener" />
13.                    <listener ref="skipCountStepExecutionListener" />
14.                </listeners>
15.            </chunk>
16.        </tasklet>
17.    </step>
18.</job>
```

使用代码清单 8-26 执行定义的 compositeJob。

完整代码参见：test.com.juxtapose.example.ch08.JobLaunchComposite。

代码清单 8-26 执行 compositeJob

```
1. JobLaunchBase.executeJob("ch08/job/job-composite.xml", "compositeJob",
2.     new JobParametersBuilder().addDate("date", new Date())
3.     .addString("filter", "true"));
```

经过部分数据处理后写入的文件内容参见代码清单 8-27。

代码清单 8-27 处理后的数据清单

```
1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road,tom
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road,tom
3. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road,tom
```

所有为消费金额为负数的行被过滤掉，每条记录的地址后面增加了名字信息。

8.6 服务复用

8.6.1 ItemProcessorAdapter

ItemProcessorAdapter 结构关键属性

ItemProcessorAdapter 持有服务对象，并调用指定的操作来完成 ItemProcessor 中定义的 process 功能。需要注意的是：已经存在的服务需要能够直接处理读提供的 Item 对象，即参数必须是读输出的 Item 的具体类型；返回值类型必须是 ItemWrite 阶段输入的参数类型。

ItemProcessorAdapter 和现有服务之间的关系参见图 8-9。

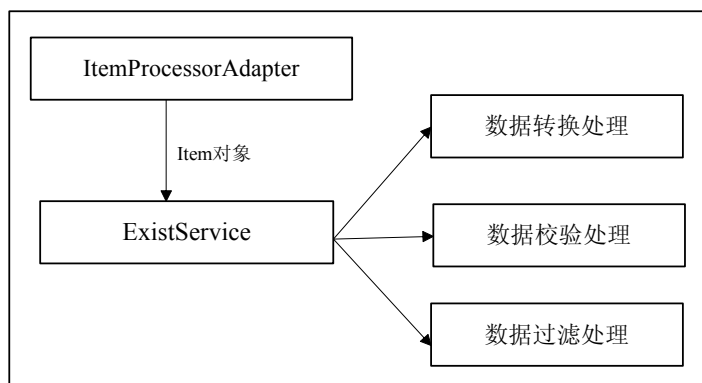


图 8-9 ItemProcessorAdapter 和现有服务之间的关系

ItemProcessorAdapter 类关系图参见图 8-10。

ItemProcessorAdapter 实现接口 ItemProcessor 并继承 AbstractMethodInvokingDelegator，后者提供了调用代理服务的一系列方法；ExistService 表示现存的服务。

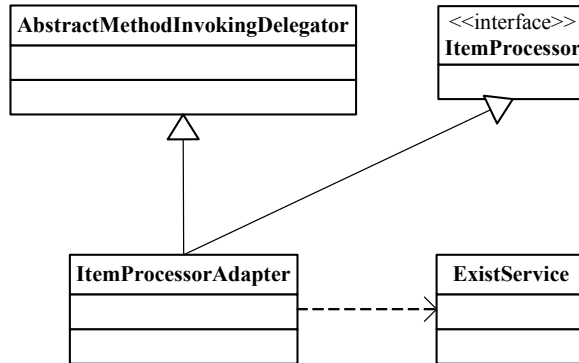


图 8-10 ItemProcessorAdapter 类关系图

ItemProcessorAdapter 关键属性

表 8-4 ItemProcessorAdapter 关键属性

ItemProcessorAdapter 属性	类 型	说 明
targetObject	Object	需要调用的目标服务对象
targetMethod	String	需要调用的目标操作名称
arguments	Object[]	需要调用的操作参数。 默认不需要传递该参数，默认情况下将每次处理的 Item 对象作为参数传入

表 8-4 给出了 ItemProcessorAdapter 的关键属性。在配置 ItemProcessorAdapter 时候只需要指定上面的前两个属性即可，参数 arguments 默认情况下将每次处理的 Item 对象作为参数传入。

配置 ItemProcessorAdapter

目前已经有服务（com.juxtapose.example.ch08.ExistService）负责对信用卡账单信息进行验证，接下来我们使用 ExistService 作为示例来演示如何使用 ItemProcessorAdapter。

已经存在的服务 ExistService 的示例代码参见代码清单 8-28。

完整代码参见：com.juxtapose.example.ch08.ExistService。

代码清单 8-28 已存在服务 ExistService 示例代码

```

1. public class ExistService {
2.     public CreditBill validate(CreditBill creditBill) throws Validation
       Exception {
3.         if(Double.compare(0, creditBill.getAmount()) >0) {
4.             throw new ValidationException("Credit bill cannot be negative!");
5.         }
  
```

```
6.         return creditBill;
7.     }
8. }
```

本服务模拟已经存在的服务，`validate` 操作负责对账单对象进行验证，不通过会抛出验证异常；通过会直接返回信用卡账单对象。

接下来我们学习如何配置 `ItemProcessorAdapter`，具体配置代码参见代码清单 8-29。

完整配置参见文件：`ch08/job/job-reuse-service.xml`。

代码清单 8-29 配置 `ItemProcessorAdapter`

```
1.     <bean:bean id="reuseServiceProcessor"
2.         class="org.springframework.batch.item.adapter.
           ItemProcessorAdapter">
3.         <bean:property name="targetObject" ref="existService"/>
4.         <bean:property name="targetMethod" value="validate"/>
5.     </bean:bean>
6.
7.     <bean:bean id="existService"
8.         class="com.juxtapose.example.ch08.ExistService"/>
```

其中，1~5 行：复用现有的服务完成 `ItemProcessorAdapter` 的配置，属性 `targetObject` 使用定义的已存服务 `existService`；属性 `targetMethod` 指定调用 `validate` 操作（参数为 `CreditBill` 的）。

7~8 行：声明现存的服务。

截止到目前我们配置完了如何使用现有的服务，通过 `ItemProcessorAdapter` 可以轻松方便地使用现有的服务功能，避免重复发明新的轮子。

8.7 拦截器

Spring Batch 框架在 `ItemProcessor` 执行阶段提供了拦截器，使得在 `ItemProcessor` 执行前后能够加入自定义的业务逻辑。`ItemProcessor` 执行阶段拦截器接口为：`org.springframework.batch.core.ItemProcessListener<T, S>`。

8.7.1 拦截器接口

接口 `ItemProcessListener` 的定义参见代码清单 8-30。

代码清单 8-30 `ItemProcessListener` 接口定义

```
1. public interface ItemProcessListener<T, S> extends StepListener {
2.     void beforeProcess(T item);
3.     void afterProcess(T item, S result);
4.     void onProcessError(T item, Exception e);
5. }
```

其中，2 行：beforeProcess 在处理操作前触发该操作。

3 行：afterProcess 在处理操作后触发该操作。

4 行：onProcessError 在处理操作发生异常的时候触发该操作。

为 ItemProcessor 配置拦截器，参见代码清单 8-31。

完整配置参见文件：/ch08/job/job-listener.xml。

代码清单 8-31 配置 ItemProcessor 拦截器

```
1.     <job id="translateJob">
2.         <step id="translateStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="flatFileItemReader" writer="partTranslate
                    FlatFileItemWriter"
5.                     processor="partTranslateItemProcessor"
6.                     commit-interval="2">
7.                         <listeners>
8.                             <listener ref="sysoutItemProcessListener"></listener>
9.                             <listener ref="sysoutAnnotationItemProcessListener">
10.                                 </listener>
11.                         </listeners>
12.                     </chunk>
13.                 </tasklet>
14.             </step>
15.         </job>
16.         <bean:bean id="sysoutItemProcessListener"
17.             class="com.juxtapose.example.ch08.listener.
18.                 SystemOutItemProcessListener">
19.         </bean:bean>
20.         <bean:bean id="sysoutAnnotationItemProcessListener"
21.             class="com.juxtapose.example.ch08.listener.SystemOutAnnotation">
22.         </bean:bean>
```

其中，6~9 行：为作业的读配置 2 个拦截器。

14~16 行：定义拦截器 sysoutItemProcessListener，该拦截器实现接口 ItemProcessListener。

18~20 行：定义拦截器 sysoutAnnotationItemProcessListener，该拦截器通过 Annotation 方式定义。

SystemOutItemProcessListener 的代码参见代码清单 8-32。

完整代码参见：com.juxtapose.example.ch08.listener.SystemOutItemProcessListener。

代码清单 8-32 SystemOutItemProcessListener 类定义

```
1. public class SystemOutItemProcessListener implements
2.     ItemProcessListener<CreditBill, CreditBill> {
3.     public void beforeProcess(CreditBill item) {
```

```

4.         System.out.println("SystemOutItemProcessListener.beforeProcess()");
5.     }
6.     public void afterProcess(CreditBill item, CreditBill result) {
7.         System.out.println("SystemOutItemProcessListener.afterProcess()");
8.     }
9.     public void onProcessError(CreditBill item, Exception e) {
10.        System.out.println("SystemOutItemProcessListener.onProcessError()");
11.    }
12. }

```

8.7.2 拦截器异常

拦截器方法如果抛出异常会影响 Job 的执行，所以在执行自定义的拦截器时候，需要考虑对拦截器发生的异常做处理，避免影响业务。

如果拦截器发生异常，会导致 Job 执行的状态为"FAILED"。

配置了错误拦截器的作业配置参见代码清单 8-33。

完整配置参见文件：/ch08/job/job-listener.xml。

代码清单 8-33 配置了错误拦截器的作业 errorTranslateJob

```

1. <job id="errorTranslateJob">
2.     <step id="errorTranslateStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="flatFileItemReader"
5.                 writer="partTranslateFlatFileItemWriter"
6.                 processor="partTranslateItemProcessor"
7.                 commit-interval="2">
8.                 <listeners>
9.                     <listener ref="errorItemProcessListener"></listener>
10.                </listeners>
11.            </chunk>
12.        </tasklet>
13.    </step>
14.</job>

```

其中，7 行：errorItemProcessListener 在 beforeProcess 操作中抛出异常，会导致整个作业的失败，当前的 Job 实例状态被标记为" FAILED "。

8.7.3 执行顺序

在配置文件中可以配置多个 ItemProcessListener，拦截器之间的执行顺序按照 listeners 定义的顺序执行。beforeProcess 方法按照 listener 定义的顺序执行，afterProcess 方法按照相反的顺序执行。上面示例代码中执行顺序如下：

1. sysoutItemProcessListener 拦截器的 beforeProcess 方法;
2. sysoutAnnotationItemProcessListener 拦截器的 beforeProcess 方法;
3. sysoutAnnotationItemProcessListener 拦截器的 afterProcess 方法;
4. sysoutItemProcessListener 拦截器的 afterProcess 方法。

8.7.4 Annotation

Spring Batch 框架提供了 Annotation 机制, 可以不实现接口 ItemProcessListener, 直接通过 Annotation 的机制定义拦截器。为 ItemProcessListener 提供的 Annotation 有:

- @BeforeProcess;
- @AfterProcess;
- @OnProcessError。

ItemProcessListener 操作说明与 Annotation 定义参见表 8-5。

表 8-5 ItemProcessListener 操作说明与 Annotation 定义

操 作	操作说明	Annotation
beforeProcess(T item)	在 ItemProcessor#process()之前执行	@ BeforeProcess
afterProcess(T item, S result)	在 ItemProcessor#process()之后执行	@ AfterProcess
onProcessError(T item, Exception e)	当 ItemProcessor#process()抛出异常时候触发该操作	@ OnProcessError

使用 Annotation 声明的拦截器的 Spring 配置文件和实现接口 ItemProcessListener 的拦截器配置一样, 只需要在 listeners 节点中声明即可。

SystemOutAnnotation 的代码参见代码清单 8-34。

完整代码参见: com.juxtapose.example.ch08.listener.SystemOutAnnotation。

代码清单 8-34 SystemOutAnnotation 类定义

```

1. public class SystemOutAnnotation {
2.     @BeforeProcess
3.     public void beforeProcess(CreditBill item) {
4.         System.out.println("SystemOutAnnotation.beforeProcess()");
5.     }
6.
7.     @AfterProcess
8.     public void afterProcess(CreditBill item, CreditBill result) {
9.         System.out.println("SystemOutAnnotation.afterProcess()");
10.    }
11.
12.    @OnProcessError

```

```

13.     public void onProcessError(CreditBill item, Exception e) {
14.         System.out.println("SystemOutAnnotation.onProcessError()");
15.     }
16. }

```

8.7.5 属性 Merge

Spring Batch 框架提供了多处配置拦截器执行,可以在 `chunk` 元素节点配置,可以在 `tasklet` 中配置;框架同样提供了 `step` 的抽象和继承的能力,可以在父 `Step` 中定义通用的属性,在子 `step` 中定义个性化的属性,通过 `merge` 属性可以定义是覆盖父中的设置、还是和父中的定义合并;`chunk` 元素中的 `listeners` 支持 `merge` 属性。

假设有这样一个场景,所有的 `Step` 都希望拦截器 `sysoutItemProcessListener` 能够执行,而拦截器 `sysoutAnnotationItemProcessListener` 则由每个具体的 `Step` 定义是否执行,通过抽象和继承属性可以完成上面的场景。

`merge` 属性配置代码参见代码清单 8-35。

代码清单 8-35 `merge` 属性配置

```

1.     <job id="mergeTranslateJob" >
2.         <step id="subChunkStep" parent="abstractParentStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="flatFileItemReader" writer="partTranslate
5.                     FlatFileItemWriter" processor="partTranslateItemProcessor"
6.                     commit-interval="2">
7.                     <listeners merge="true">
8.                         <listener ref="sysoutAnnotationItemProcessListener">
9.                             </listener>
10.                        </listeners>
11.                    </chunk>
12.                </tasklet>
13.            </step>
14.        </job>
15.
16.        <step id="abstractParentStep" abstract="true">
17.            <tasklet>
18.                <chunk commit-interval="2" >
19.                    <listeners>
20.                        <listener ref="sysoutItemProcessListener"></listener>
21.                    </listeners>
22.                </chunk>
23.            </tasklet>
24.        </step>

```

其中，17 行：定义抽象作业步 `abstractParentStep` 中的拦截器。

5~7 行：通过 `merge` 属性，可以与父类中的拦截器配置进行合并，表示在 `subChunkStep` 中有两个拦截器会同时工作；属性 `merge` 的值为 `true`，表示父子合并即两处定义的都生效；实行 `merge` 的值为 `false`，表示父类的不再生效，被子类的定义覆盖掉。

通过 `merge` 属性合并的拦截器的执行顺序如下：首先执行父 `Step` 中定义的拦截器，然后执行子 `Step` 中定义的拦截器。

第 3 篇 高级篇

本篇重点讲述了批处理的高性能、高可靠性和并行处理的能力，分别向读者展示：如何实现作业流的控制包括顺序流、条件流、并行流，如何实现健壮的作业包括跳过、重试、重启等，如何实现扩展作业及并行作业包括多线程作业、并行作业、远程作业、分区作业等。本篇包含三个章节。

第 9 章：介绍作业流，包括顺序 Flow、条件 Flow、并行 Flow、外部 Flow，最后向读者介绍了作业步 Step 之间如何数据共享和多种终止 Job 的方式，包括 end、stop、fail 三种方式。

第 10 章：介绍如何设计健壮的作业 Job，主要包括跳过 Skip、重试 Retry、重启 Restart 三种可靠的策略。

第 11 章：介绍如何设计高可靠、高性能、支持大并发处理的作业，重点向读者介绍了 Spring Batch 框架提供的四种扩展策略：多线程 Step、并行 Step、远程 Step 和分区 Step。

作业流 Step Flow

前面章节所有的示例中，每个 Job 中只有一个 Step。Spring Batch 框架支持一个 Job 中配置多个 Step，不同的 Step 之间可以顺序执行，也可以按照不同的条件有选择地执行（条件通常使用 Step 的退出状态决定），通过 next 元素或者 decision 元素来定义跳转规则；为了提高多个 Step 的执行效率，框架提供了 Step 并行执行的能力（通常该情况下需要 Step 之间没有任何的依赖关系，否则容易引起业务上的错误）。

批处理任务中有些任务有先后的执行顺序，还有些 Step 没有先后执行顺序的要求，可以同一时刻并行作业。批处理框架提供了并行处理 Step 的能力，通过 Split 元素可以定义并行的作业流，提高 Job 的执行效率。

除了正常完成 Step 后终止作业的执行，Spring Batch 框架提供了其他终止 Job 的能力，通过 End、Stop、Fail 等元素可以在任意需要终止 Job 的 Step 中终止作业的执行。

为了提高 Step 的复用，Spring Batch 框架提供了 flow（作业流）、FlowStep、JobStep 的支持，三者类似 Java 领域中的继承和复用的功能，使得 Job、Flow、Step 可以复用。

不同的 Step 之间如果需要共享数据，通过执行上下文 ExecutionContext，可以在不同的作业步 Step 之间达到共享数据的目的。

9.1 顺序 Flow

顺序 Flow 是指在 Job 中定义多个 Step，每个 Step 之间按照定义好的顺序执行，任何一个 Step 的失败都会导致 Job 的失败。元素 step 提供了 next 属性，可以定义当前 Step 执行完毕后下一个需要执行的 Step。顺序 Flow 执行的示意图参见图 9-1，表示作业步 StepA、StepB、StepC 三者顺序执行。

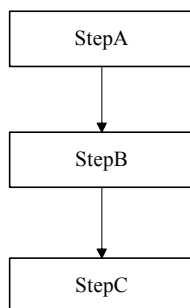


图 9-1 顺序 Flow 示意图

Step 元素 Schema 的定义参见图 9-2。

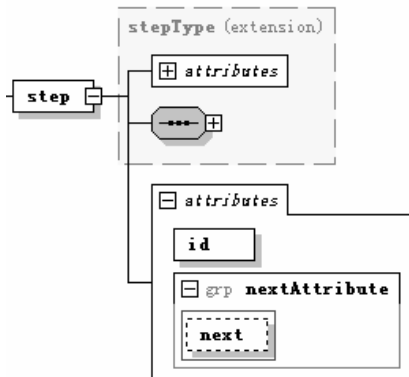


图 9-2 Step 元素 Schema 定义

next 属性定义当前 Step 执行完毕后接下来需要执行的 Step。

接下来使用信用卡账单处理的示例展示如何开发顺序的 Job。前面所有的章节直接读入 csv 格式的文件进行处理，在实际的企业应用中，账单文件通常以压缩的方式存放传输，批处理应用接收到的是压缩的文件，首选需要进行解压缩；解压缩后需要对文件是否存在，文件名称是否正确等进行校验；校验通过后才真正执行文件的解析，读取，数据保存等工作；最后需要将处理过程遗留下的临时文件清除，恢复工作环境。账单处理步骤参见图 9-3。

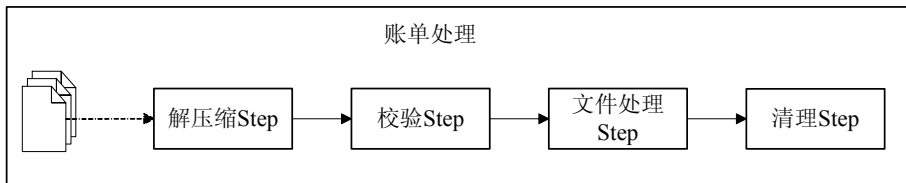


图 9-3 账单处理步骤

账单处理作业共有 4 个作业步，解压缩 Step、校验 Step、文件处理 Step 和清理 Step。配置顺序的作业步参见代码清单 9-1，在 Step 中通过 next 属性指定后续需要处理的 Step。

完整配置参见文件：ch09/job/job-sequential.xml。

代码清单 9-1 配置顺序的作业步

```
1.     <job id="sequentialJob" >
2.         <step id="decompressStep" parent="abstractDecompressStep"
3.             next="verifyStep" >
4.             <tasklet ref="decompressTasklet" />
5.         </step>
6.         <step id="verifyStep" next="readWriteStep">
7.             <tasklet ref="verifyTasklet" />
```

```

8.         </step>
9.         <step id="readWriteStep" parent="parentReadWriteStep"
10.             next="cleanStep" />
11.         <step id="cleanStep">
12.             <tasklet ref="cleanTasklet" />
13.         </step>
14.     </job>

```

其中，2~5 行：定义 decompressStep（解压缩 Step），完成文件的解压缩功能，next 属性定义了下一个作业步为校验 Step。

5~8 行：定义 verifyStep（校验 Step），完成对解压缩后文件的校验功能（文件是否存在，是否可读等），next 属性定义了下一个作业步为真正的文件读/写 Step。

9~10 行：定义 readWriteStep（文件读/写 Step），完成信用卡账单文件的读、处理、写的功能，next 属性定义了下一个作业步为清理 Step。

11~13 行：定义 cleanStep（清理 Step），完成临时目录的清理。

说明：顺序定义的 Step 中，执行顺序按照 Job 中定义的顺序执行，因此上面第一个执行的是 decompressStep。

9.2 条件 Flow

更多的业务场景需要根据作业步的执行结果决定后续调用哪个作业步，而不是像上节预先就定义好了作业的执行顺序。Spring Batch 框架提供了条件 Flow 来满足有选择的执行作业步的功能。

条件 Flow 执行示意图参见图 9-4，作业步 StepA 执行完毕后根据条件判断可能执行 StepB、也可能执行 StepC。

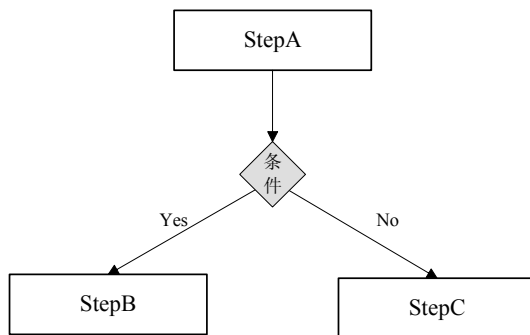


图 9-4 条件 Flow 示意图

条件 Flow 是指在 Job 中定义多个 Step，每个 Step 之间按照条件定义的规则执行。可以使用 Step 元素中的 next 元素定义条件；或者使用 Job 的子元素 decision 来定义跳转条件。

9.2.1 next

Step 中 next 元素的 Schema 定义参见图 9-5。

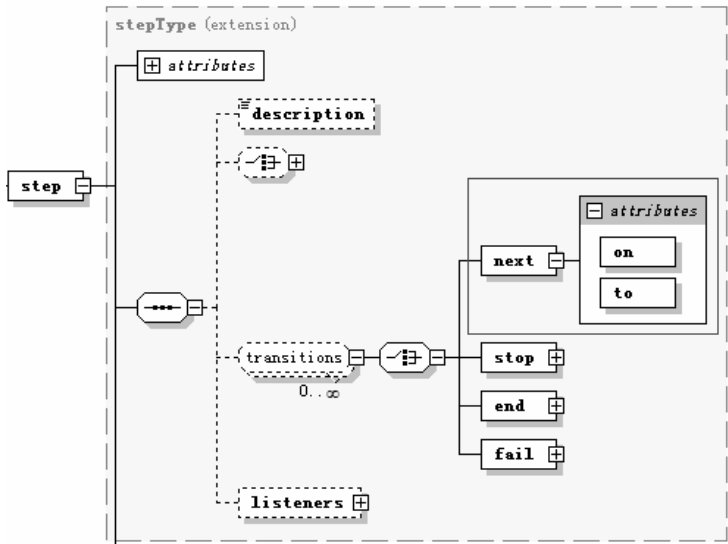


图 9-5 Step 中 next 元素 Schema 定义

next 属性说明参见表 9-1。

表 9-1 next 属性说明

属 性	说 明	默 认 值
on	定义作业步的 ExitStatus（退出状态）和 on 属性指定的值匹配的时候，则执行 to 指定的作业步。 on 属性的值可以是任意的字符串，同时支持通配符"*"、"?" "*"：表示退出状态为任何值都满足。 "?"：表示匹配一个字符，如 c?t，当作业的退出状态为 cat 的时候满足，如果是 cabt 则不满足	
to	当前 Step 执行完成后，to 属性元素指定下个需要执行的 Step	

使用上节中的示例，在校验 Step 环节增加条件判断，如果输入的文件满足条件则进入文件处理的 Step，否则跳过文件处理 Step，直接进入清理 Step 中。

条件账单处理步骤参见图 9-6。

配置条件作业步，参见代码清单 9-2，校验 Step 中增加了条件判断，通过 on 属性匹配作业步 Step 的退出状态，当 on 属性定义的值与 Step 退出状态匹配时，执行属性 to 对应的 Step。

完整配置参见文件：ch09/job/job-conditional.xml。

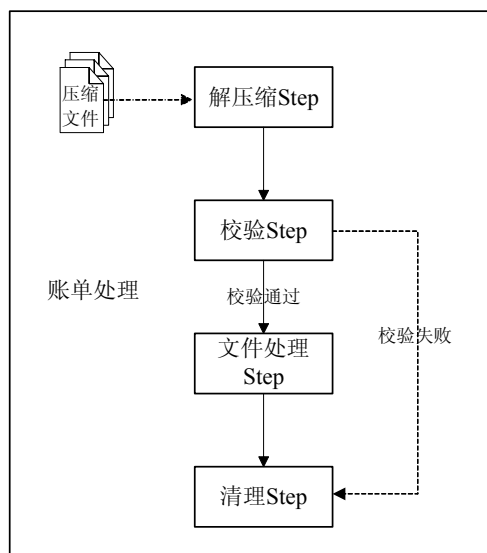


图 9-6 条件账单处理步骤

代码清单 9-2 配置条件作业步

```

1.     <job id="conditionalJob" >
2.         <step id="decompressStep" parent="abstractDecompressStep"
3.             next="verifyStep" >
4.             <tasklet ref="decompressTasklet" />
5.         </step>
6.         <step id="verifyStep" >
7.             <tasklet ref="verifyTasklet" />
8.             <next on="*" to="readWriteStep" />
9.             <next on="SKIPTOCLEAN" to="cleanStep" />
10.            <listeners>
11.                <listener ref="verifyStepExecutionListener" />
12.            </listeners>
13.        </step>
14.        <step id="readWriteStep" parent="parentReadWriteStep"
15.            next="cleanStep" />
16.        <step id="cleanStep">
17.            <tasklet ref="cleanTasklet" />
18.        </step>
19.    </job>
  
```

其中,8行:属性 on 定义为"*,表示当前作业步 verifyStep 的退出状态非"SKIPTOCLEAN"的情况下都会跳转到 readWriteStep 中。

9行:属性 on 定义为"SKIPTOCLEAN",表示步 verifyStep 的返回值为"SKIPTOCLEAN"时,会跳转到作业步 cleanStep 中。

10~12 行：声明作业步执行器，通过该拦截器修改 Step 的退出状态。

说明：属性 on 中定义的值，是 Step 的 ExitStatus 退出状态，可以通过拦截器 StepExecutionListener 修改 Step 的退出状态的值。

通过 Step 拦截器修改作业步 verifyStep 的退出状态，通常在拦截器 StepExecutionListener 的 afterStep() 操作中修改退出状态的值。

拦截器 VerifyStepExecutionListener 的实现参见代码清单 9-3，在该拦截器中显式修改了 Step 的退出状态。

完整代码参见：com.juxtapose.example.ch09.listener.VerifyStepExecutionListener。

代码清单 9-3 VerifyStepExecutionListener 类定义

```
1. public class VerifyStepExecutionListener implements StepExecutionListener {
2.     public void beforeStep(StepExecution stepExecution) { }
3.     public ExitStatus afterStep(StepExecution stepExecution) {
4.         String status = stepExecution.getExecutionContext()
5.             .getString(Constant.VERITY_STATUS);
6.         if(null != status){
7.             return new ExitStatus(status);
8.         }
9.         return stepExecution.getExitStatus();
10.    }
11. }
```

其中，4~5 行：从作业步上下文中获取退出状态的值，如果该值存在则返回；作业步上下文在整个作业执行的声明周期中生效，本例在作业步 verifyStep 的实现 verifyTasklet（参见代码清单 9-4）中将作业步的状态放入上下文中，然后在拦截器中修改。

9 行：如果作业步上下文中没有获取到，则直接返回当前作业步的退出状态值。

作业步 verifyStep 的实现代码参见代码清单 9-4。

完整代码参见：com.juxtapose.example.ch09.tasklet.VerifyTasklet。

代码清单 9-4 VerifyTasklet 类定义

```
1. public class VerifyTasklet implements Tasklet {
2.     public RepeatStatus execute(StepContribution contribution,
3.         ChunkContext chunkContext) throws Exception {
4.         String status = creditService.verify(outputDirectory, readFileName);
5.         if (null != status) {
6.             chunkContext.getStepContext().getStepExecution()
7.                 .getExecutionContext().put(Constant.VERITY_STATUS, status);
8.         }
9.         return RepeatStatus.FINISHED;
10.    }
11. }
```

其中，4 行：调用服务获取校验的状态信息。

5~8 行：将状态值放入当前的作业步上下文中。

通过 `next` 元素可以方便地设置条件 `Flow`，`on` 属性指定退出状态的匹配值，`to` 属性指定当前 `Step` 完成后下一个执行的 `Step`。

9.2.2 ExitStatus VS BatchStatus

Spring Batch 框架有两个重要的状态，一个是 9.2.1 节提到的退出状态（`ExitStatus`），另一个是批处理状态（`BatchStatus`）。本节向读者分别介绍这两个状态的差异。

9.2.2.1 BatchStatus

批处理状态通常由批处理框架使用，用来记录 `Job` 或者 `Step` 的执行情况，在 `Job` 或者 `Step` 的重启中，批处理的状态起到关键作用（`Job` 或者 `Step` 的重启状态判断使用的就是 `BatchStatus`）。可以通过 `Job Execution` 和 `Step Execution` 获取当前 `Job`、`Step` 的批处理状态。

`JobExecution.getStatus()`操作可以获取作业 `Job` 的批处理状态。

`StepExecution.getStatus()`操作可以获取作业步 `Step` 的批处理状态。

批处理的状态是枚举类型，目前框架定义了 8 种类型，分别是完成 `COMPLETED`、启动中 `STARTING`、已启动 `STARTED`、停止中 `STOPPING`、已停止 `STOPPED`、失败 `FAILED`、废弃 `ABANDONED` 和未知 `UNKOWN`。

批处理状态属性列表参见表 9-2。

表 9-2 批处理状态属性

状 态	说 明
COMPLETED	表示完成状态，所有的 <code>Step</code> 都标记为 <code>COMPLETED</code> 后， <code>Job</code> 会处于此状态
STARTING	表示作业正在启动状态，还没有启动完毕
STARTED	表示作业已经成功启动
STOPPING	表示作业正在停止中
STOPPED	表示作业停止完成
FAILED	表示作业执行失败
ABANDONED	表示当前下次重启 <code>Job</code> 时候需要废弃掉的 <code>Step</code> ，即不会被再次执行
UNKOWN	表示未知的状态，该状态下重启 <code>Job</code> 会抛出错误

批处理状态在 `Job` 或者 `Step` 执行期间通过 `Job` 上下文或者 `Step` 上下文持久化到 `DB` 中，可以在表 `batch_job_execution` 查看 `Job` 的批处理状态（字段 `STATUS` 表示该状态），在表 `batch_step_execution` 查看 `Step` 的批处理状态（字段 `STATUS` 表示该状态）。

`Job` 的批处理状态在表 `batch_job_execution` 中的示例，参见图 9-7。

	JOB_EXECUTION_ID	STATUS	VERSION	JOB_INSTANCE_ID
1	152	STOPPED	2	151
2	153	COMPLETED	2	151
3	154	COMPLETED	2	152

图 9-7 Job 批处理状态示例

Step 的批处理状态在表 batch_step_execution 中的示例，参见图 9-8。

	STEP_EXECUTION_ID	STATUS	VERSION	STEP_NAME	JOB_EXECUTION_ID
1	221	COMPLETED	3	decompressStep	152
2	222	COMPLETED	3	verifyStep	152
3	223	COMPLETED	6	readWriteStep	153
4	224	COMPLETED	3	cleanStep	153
5	225	COMPLETED	3	decompressStep	154
6	226	COMPLETED	3	verifyStep	154
7	227	COMPLETED	6	readWriteStep	154
8	228	COMPLETED	3	cleanStep	154

图 9-8 Step 批处理状态示例

9.2.2.2 ExitStatus

退出状态表示 Step 执行后或者 Job 执行后的状态，该状态值可以被修改，通常用于条件 Flow 中。可以通过拦截器 StepExecutionListener 的 afterStep()操作来修改退出状态的值。

StepExecutionListener 接口声明参见代码清单 9-5。

代码清单 9-5 StepExecutionListener 接口定义

```

1. public interface StepExecutionListener extends StepListener {
2.     void beforeStep(StepExecution stepExecution);
3.     ExitStatus afterStep(StepExecution stepExecution);
4. }
```

其中，2 行：表示在 Step 执行之前调用该方法。

3 行：表示在 Step 执行之后调用该方法，读者需要注意，该操作的返回值是 ExitStatus，表示当次作业步执行后的退出状态；可以实现该接口修改 ExitStatus 的值。

如何通过拦截器设置退出状态的值可以参见 9.2.1 节的示例。

退出状态在 Job 或者 Step 执行期间通过 Job 上下文或者 Step 上下文持久化到 DB 中，可以在表 batch_job_execution 查看 Job 的退出状态（字段 EXIT_CODE 表示该状态），在表 batch_step_execution 查看 Step 的退出状态（字段 EXIT_CODE 表示该状态）。

Job 的退出状态在表 batch_job_execution 中的示例参见图 9-9。

	JOB_EXECUTION_ID	STATUS	EXIT_CODE	VERSION
1	152	STOPPED	STOPPED	2
2	153	COMPLETED	COMPLETED	2
3	154	COMPLETED	COMPLETED	2

图 9-9 Job 退出状态示例

Step 的退出状态在表 batch_step_execution 中的示例参见图 9-10。

	STEP_EXECUTION_ID	STATUS	EXIT_CODE	VERSION	STEP_NAME
1	221	COMPLETED	COMPLETED	3	decompressStep
2	222	COMPLETED	COMPLETED	3	verifyStep
3	223	COMPLETED	COMPLETED	6	readWriteStep
4	224	COMPLETED	COMPLETED	3	cleanStep
5	225	COMPLETED	COMPLETED	3	decompressStep
6	226	COMPLETED	COMPLETED	3	verifyStep
7	227	COMPLETED	COMPLETED	6	readWriteStep
8	228	COMPLETED	COMPLETED	3	cleanStep

图 9-10 Step 退出状态示例

9.2.3 decision 条件

Spring Batch 框架提供了 decision 支持条件 Flow。decision 是 Job 的子元素，所有 decision 的实现必须实现接口 JobExecutionDecider。

元素 decision 的 Schema 定义参见图 9-11。

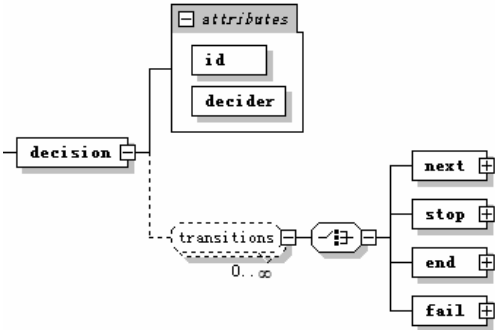


图 9-11 decision 的 Schema 定义

decision 属性说明参见表 9-3，核心属性为 id、decider。

表 9-3 decision 属性说明

属 性	说 明	默 认 值
id	Job 定义中的唯一 ID	
decider	条件决定器的实现，需要实现接口 JobExecutionDecider	

Decision 中可以定义子元素 next, stop, end, fail 等，用于进行跳转的具体定义工作；如何使用 stop, end, fail 请参见 9.6 节的描述，next 元素参见 9.2.1 节描述。

JobExecutionDecider 接口

JobExecutionDecider 接口声明参见代码清单 9-6。

接口全称：org.springframework.batch.core.job.flow.JobExecutionDecider。

代码清单 9-6 JobExecutionDecider 接口定义

```
1. public interface JobExecutionDecider {
2.     FlowExecutionStatus decide(JobExecution jobExecution, StepExecution
        stepExecution);
3. }
```

其中，2 行：接口唯一方法，用来决定如何跳转，参数是作业执行器和作业步执行器；返回值为 FlowExecutionStatus，可以使用自带的枚举值，也可以自定义自己的 FlowExecutionStatus。

本节仍然使用信用卡账单处理的场景，整个作业需要四个作业步，分别是：解压缩 Step、校验 Step、文件处理 Step、清理 Step。为了提高 Job 处理的效率，需要在解压缩 Step 之后判断解压缩的文件是否存在，不存在直接跳转到清理 Step，存在则跳转到校验 Step、文件处理 Step。

账单处理场景示意图参见图 9-12。

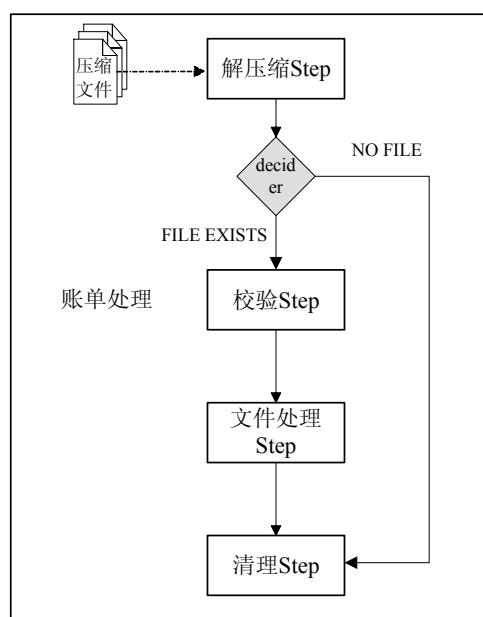


图 9-12 条件账单处理步骤

接下来我们首先实现文件是否存在 FileExistsDecider 类（代码实现参见代码清单 9-7），FileExistsDecider 实现接口 JobExecutionDecider，根据文件是否存在返回特定的退出状态标识，如果文件存在返回“FILE EXISTS”，不存在返回“NO FILE”。

完整代码参见：com.juxtapose.example.ch09.decider.FileExistsDecider。

代码清单 9-7 FileExistsDecider 类定义

```
1. public class FileExistsDecider implements JobExecutionDecider {
2.     private CreditService creditService;
3.     public FlowExecutionStatus decide(JobExecution jobExecution,
```

```

4.         StepExecution stepExecution) {
5.         String readFileName = stepExecution.getJobParameters()
6.                                     .getString(Constant.READ_FILE_NAME);
7.         String workDirectory = stepExecution.getJobParameters()
8.                                     .getString(Constant.WORK_DIRECTORY);
9.         if(creditService.exists(workDirectory+readFileName)) {
10.            return new FlowExecutionStatus("FILE EXISTS");
11.        } else {
12.            return new FlowExecutionStatus("NO FILE");
13.        }
14.    }
15.    .....
16. }

```

其中，10 行：文件存在返回退出状态标识为“FILE EXISTS”。

12 行：文件不存在返回退出状态标识为“NO FILE”。

最后我们来配置 decision 的 Job，参见代码清单 9-8。根据类 FileExistsDecider 的返回退出状态标识决定 Step 的流转，如果返回标识为“FILE EXISTS”则执行校验 Step，如果返回标识为“NO FILE”则执行清理 Step。

完整配置参见文件：ch09/job/job-conditional-decider.xml。

代码清单 9-8 配置 decision 的 Job

```

1.     <job id="conditionalDeciderJob" >
2.         <step id="decompressStep" parent="abstractDecompressStep"
3.             next="decision" >
4.             <tasklet ref="decompressTasklet" />
5.         </step>
6.
7.         <decision id="decision" decider="fileExistsDecider">
8.             <next on="FILE EXISTS" to="verifyStep" />
9.             <next on="NO FILE" to="cleanStep" />
10.        </decision>
11.
12.        <step id="verifyStep" >
13.            <tasklet ref="verifyTasklet" />
14.            <next on="*" to="readWriteStep" />
15.            <next on="SKIPTOCLEAN" to="cleanStep" />
16.        </step>
17.
18.        <step id="readWriteStep" parent="parentReadWriteStep"
19.            next="cleanStep" />
20.
21.        <step id="cleanStep">
22.            <tasklet ref="cleanTasklet" />

```

```

22.         </step>
23.     </job>
24.
25.     <bean:bean id="fileExistsDecider"
26.         class="com.juxtapose.example.ch09.decider.FileExistsDecider">
27.         <bean:property name="creditService" ref="creditService" />
28.     </bean:bean>

```

其中，1~3 行：作业步 decompressStep 属性 next 直接指向 fileExistsDecider，表示作业步 decompressStep 执行成功后，会跳转到 fileExistsDecider，由 fileExistsDecider 决定后续的作业步。

7~10 行：定义 decision，属性 decider 定义条件决定器的实现为 fileExistsDecider。

25~28 行：定义条件决定器 fileExistsDecider 的具体实现。

9.3 并行 Flow

批处理任务中有些任务有先后的执行顺序，还有些 Step 没有先后执行顺序的要求，可以在同一时刻并行作业，批处理框架提供了并行处理 Step 的能力，通过 Split 元素可以定义并行的作业流，为 split 定义执行的线程池，从而提高 Job 的执行效率。

Spring Batch 框架提供了 split 元素来执行并行作业的能力。

元素 split 的 Schema 定义参见图 9-13。

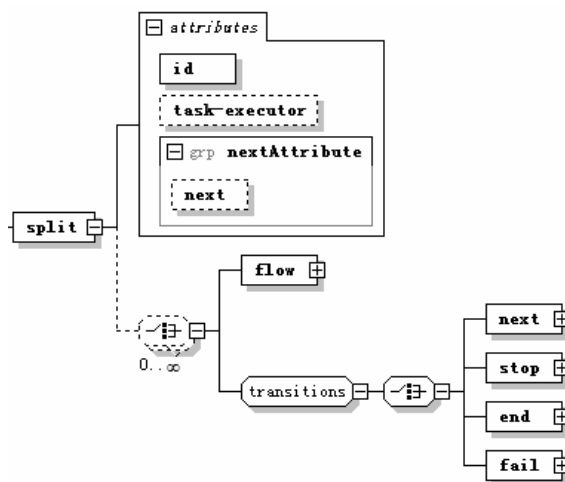


图 9-13 元素 split 的 Schema 定义

split 元素的属性说明参见表 9-4。

split 元素说明参见表 9-5。split 可以定义子元素 next，stop，end，fail 等，用于进行跳转的具体定义工作；如何使用 stop，end，fail 请参见 9.6 节的描述；split 的子元素 flow 用来定义并行处理的作业，并列的 flow 表示可以并行处理的任务。

表 9-4 split 元素的属性说明

属 性	说 明	默 认 值
id	定义 split 的唯一 ID，全局需要保证 id 唯一	
task-executor	任务执行处理器，定义后表示采用多线程执行任务，需要考虑多线程执行任务时候的安全性	如果不定义的话，默认使用同步线程执行器： SyncTaskExecutor
next	当前 split 中所有的 flow 执行完毕后，接下来执行的 Step	

表 9-5 split 元素说明

属 性	说 明
flow	用来定义并行处理的作业，并列的 flow 表示可以并行处理的任务；split 元素下面可以定义多个 flow 节点
next	根据退出状态定义下一步需要执行的 Step
stop	根据退出状态决定是否退出当前的任务，同时 Job 也会停止，作业状态为"STOPPED"
fail	根据退出状态决定是否失败当前的任务，同时 Job 也会停止，作业状态为" FAILED"
end	根据退出状态决定是否结束当前的任务，同时 Job 也会停止，作业状态为" COMPLETED"

本节仍然使用信用卡账单处理的场景，现在需要同时处理 10、11 月份的账单，处理两个月份的账单任务没有依赖关系因此可以并行处理。接下来我们并行完成 10、11 月份的账单的处理，两个任务都完成后继续执行清理的 Step，具体参见图 9-14。

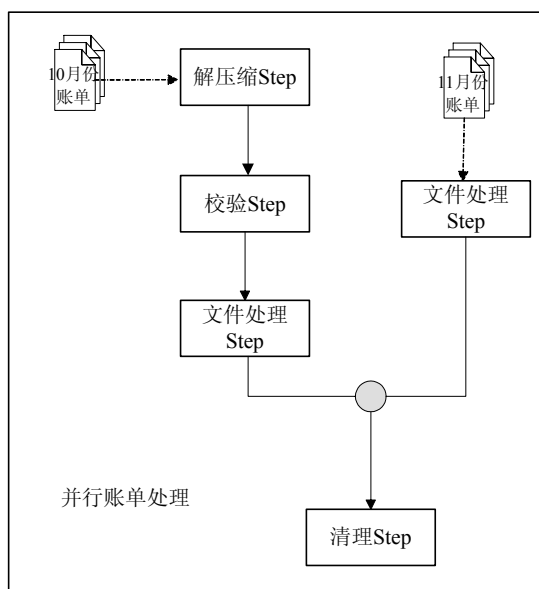


图 9-14 并行场景示意图

配置 split 的 Job 参见代码清单 9-9。

完整配置参见文件：ch09/job/job-split.xml。

代码清单 9-9 配置 split 的 Job

```
1.     <job id="splitJob" >
2.         <split id="split" task-executor="taskExecutor" next="cleanStep">
3.             <flow>
4.                 <step id="decompressStep" parent="abstractDecompressStep"
5.                     next="verifyStep" >
6.                     <tasklet ref="decompressTasklet" />
7.                 </step>
8.                 <step id="verifyStep" next="readWrite_10Step">
9.                     <tasklet ref="verifyTasklet" />
10.                </step>
11.                <step id="readWrite_10Step" parent="parentReadWriteStep"/>
12.            </flow>
13.            <flow>
14.                <step id="readWrite_11Step" parent="parentReadWriteStep">
15.                    <listeners>
16.                        <listener ref="splitStepExecutionListener"></listener>
17.                    </listeners>
18.                </step>
19.            </flow>
20.        </split>
21.        <step id="cleanStep">
22.            <tasklet ref="cleanTasklet" />
23.        </step>
24.    </job>
```

其中，2 行：使用 split 元素定义并行处理的作业步，属性 task-executor 用来定义执行多个 flow 时候使用的线程池，属性 next 定义当前的 split 执行完毕后接下来执行的 step 任务。

3~12 行：定义处理 10 月份账单的作业 flow，该 flow 定义了 3 个作业步，分别是 decompressStep、verifyStep、readWrite_10Step。

13~19 行：定义处理 11 月份账单的作业 flow，在 Job 执行期间和上面的 flow 可以并行执行。

21~23 行：定义 split 执行完毕后接下来执行的 cleanStep。

说明：split 内部的 flow 之间是并行执行的，只有当 split 内部所有的 flow 执行完毕后，才会继续执行 next 属性指定的 Step。可以为 split 指定执行的线程池。

接下来我们定义 split 使用的线程池 task-executor，具体参见代码清单 9-10。

代码清单 9-10 配置 split 使用的线程池

```
1.     <bean:bean id="taskExecutor"
2.         class="org.springframework.scheduling.concurrent.
```

```

        ThreadPoolTaskExecutor">
3.     <bean:property name="corePoolSize" value="5"/>
4.     <bean:property name="maxPoolSize" value="15"/>
5.     </bean:bean>

```

其中，3 行：属性 `corePoolSize` 定义线程池的初始大小为 5。

4 行：属性 `maxPoolSize` 定义线程池的最大值为 15。

通过属性 `task-executor` 指定对应的线程池，并行处理的作业步之间使用线程池中线程进行作业。

9.4 外部 Flow 定义

Spring Batch 框架提供了外部定义 Flow 的能力,通过在 Job 外部定义 Flow 可以做到复用。框架本身提供了三种方式的 Flow 复用，分别是：

- (1) 作业引用外部 Flow；
- (2) 作业步引用外部 Flow，称之为 `FlowStep`；
- (3) 作业步引用外部定义的 Job，称之为 `JobStep`。

接下来我们学习这三种外部 Flow 的使用方式。

9.4.1 Flow

Spring Batch 框架提供了 Flow 元素用于声明一组 Step，使用 Flow 元素声明的作业流可以被 Job 直接使用。就相当于在 Job 内部定义了一组 Step。

元素 flow 的 Schema 定义参见图 9-15。

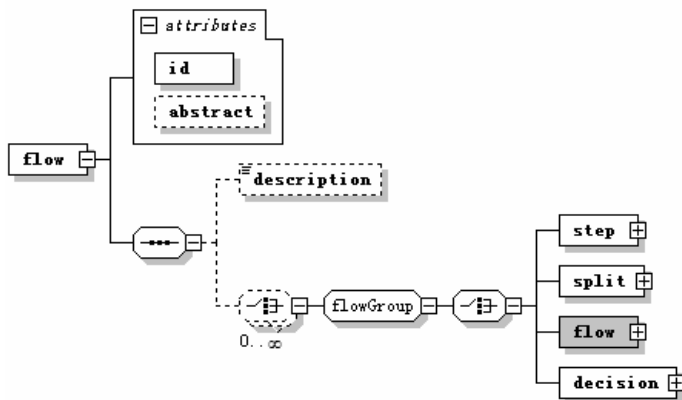


图 9-15 元素 flow 的 Schema 定义

flow 元素的属性说明参见表 9-6。

表 9-6 flow 元素的属性说明

属 性	说 明	默 认 值
id	定义 flow 的唯一 ID，全局需要保证 id 唯一	
abstrace	当前 flow 是否是抽象的	false

flow 的子元素可以是 step、split、flow、decision 四种。step 用于直接定义一个作业步；split 用于定义并行的作业步；flow 用于定义一批作业流；decision 用于进行作业的条件判断。

一个典型的 Flow 的定义参见代码清单 9-11。

完整配置文件参见：ch09/job/job-external-flow.xml。

代码清单 9-11 典型的 Flow 定义示例

```

1.    <flow id="flow1">
2.        <step id="decompressStep" parent="abstractDecompressStep"
          next="verifyStep" >
3.            <tasklet ref="decompressTasklet" />
4.        </step>
5.        <step id="verifyStep" next="readWriteStep">
6.            <tasklet ref="verifyTasklet" />
7.        </step>
8.        <step id="readWriteStep" parent="parentReadWriteStep"/>
9.    </flow>

```

定义当前的 flow 的 id 为“flow1”，内部定义了三个作业步，分别是"decompressStep"、"verifyStep"、"readWriteStep"。

接下来描述 Job 中 flow 元素的 Schema 定义，参见图 9-16。

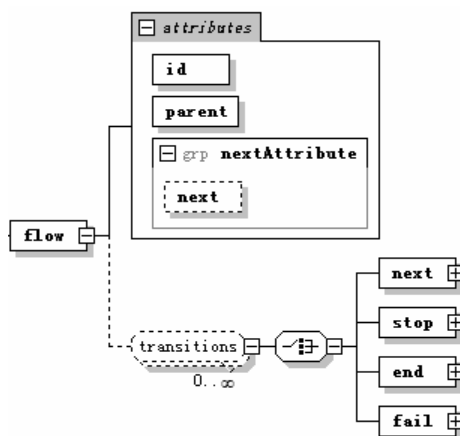


图 9-16 flow 元素的 Schema 定义

job 中 flow 元素的属性说明，参见表 9-7。

表 9-7 job 中 flow 元素的属性说明

属 性	说 明	默 认 值
id	定义 flow 的唯一 ID，全局需要保证 id 唯一	
parent	用于指定外部定义的 flow 的 ID	
next	当前 flow 中所有的 step 执行完毕后，接下来执行的 step	

接下来向读者给出如何在 Job 中使用 flow 来配置作业，具体参见代码清单 9-12。

完整配置文件参见：ch09/job/job-external-flow.xml。

代码清单 9-12 Job 引用外部 Flow

```

1.     <job id="externalFlowJob" >
2.         <flow id="externalFlowJob.flow1" parent="flow1" next="cleanStep"/>
3.         <step id="cleanStep">
4.             <tasklet ref="cleanTasklet" />
5.         </step>
6.     </job>

```

其中，2 行：使用 flow 定义 Job 的作业流，使用 parent 属性指定使用 id 为"flow1"的外部定义。

本处使用 flow 元素，在执行过程中相当于将 flow 的定义在此处复制了一份；上面的配置和代码清单 9-13 完成的功能完全一致。

代码清单 9-13 Job 引用 flow 的等价配置

```

1.     <job id="externalFlowJob" >
2.         <step id="decompressStep" parent="abstractDecompressStep"
3.             next="verifyStep" >
4.             <tasklet ref="decompressTasklet" />
5.         </step>
6.         <step id="verifyStep" next="readWriteStep">
7.             <tasklet ref="verifyTasklet" />
8.         </step>
9.         <step id="readWriteStep" parent="parentReadWriteStep"/>
10.        <step id="cleanStep">
11.            <tasklet ref="cleanTasklet" />
12.        </step>
13.    </job>

```

本处例子直接使用 step 定义，和在外部使用 flow 定义没有任何区别，唯一的好处在于，通过 flow 可以进行 Step 的组装，提供较好的服务复用功能。

执行使用 flow 定义的 Job，查看数据库的源信息可以看到，作业 externalFlowJob 先后执行了 4 个 Step。（执行代码入口：test.com.juxtapose.example.ch09.JobLaunchExternalFlow）

作业实例表信息，参见图 9-17。

	JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
1	154	0	externalFlowJob	55b94c2720e510eaf4e9bd9435c85dc

图 9-17 作业实例表信息

当前的 Job Instance 的 ID 为 154，作业名字为 externalFlowJob。

作业执行器表信息，参见图 9-18。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	CREATE_TIME
1	156	2	154	2013-10-14 22:09:21

图 9-18 作业执行器表信息

当前作业执行器的 ID 为 156，对应的 Job Instance 为 154；

作业步执行器表信息，参见图 9-19。

	STEP_EXECUTION_ID	VERSION	STEP_NAME	JOB_EXECUTION_ID
1	234	3	decompressStep	156
2	235	3	verifyStep	156
3	236	6	readWriteStep	156
4	237	3	cleanStep	156

图 9-19 作业步执行器表信息

分别执行了四个作业步，对应的作业执行器的 ID 为 156。

9.4.2 FlowStep

FlowStep 是指 Step 元素使用外部定义的 Flow，FlowStep 和直接使用 Flow 的区别在于前者将 Flow 作为一个单独的 Step 在 Job 执行期间出现（该单独的 Step 是将 Flow 中定义的所有 Step 包装在一个大的 Step 中）；后者没有单独的 Step，Flow 中定义了几个 Step，则执行期间有多少 Step。

接下来我们给出 FlowStep 的示例参见代码清单 9-14。

完整配置参见文件：ch09/job/job-external-flow-step.xml。

代码清单 9-14 FlowStep 示例

```

1.     <job id="externalFlowStepJob" >
2.         <step id="externalFlowStepJob.flow1" next="cleanStep">
3.             <flow parent="flow1" />
4.         </step>
5.         <step id="cleanStep">
6.             <tasklet ref="cleanTasklet" />
7.         </step>
8.     </job>
9.
10.    <flow id="flow1">
11.        <step id="decompressStep" parent="abstractDecompressStep"

```

```

        next="verifyStep" >
12.         <tasklet ref="decompressTasklet" />
13.     </step>
14.     <step id="verifyStep" next="readWriteStep">
15.         <tasklet ref="verifyTasklet" />
16.     </step>
17.     <step id="readWriteStep" parent="parentReadWriteStep"/>
18. </flow>

```

Step Flow 的配置是使用 Step，Step 的实现内嵌了外部定义的 flow。

使用 FlowStep 的好处在于，在 Job 中将外部定义的 Flow 作为一个完整的 Step；同时 Flow 中定义的多个 Step 在 Job 执行期间作为完整 Step 的一个子活动，当 Flow 中所有的 Step 执行完毕后 FlowStep 才会执行完成。

执行使用 FlowStep 定义的 Job，查看数据库的源信息可以看到，作业 externalFlowStepJob 先后执行了 4 个 Step。（执行代码入口：test.com.juxtapose.example.ch09.JobLaunchExternalFlowStep）作业实例表信息参见图 9-20。

	JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
1	158	0	externalFlowStepJob	6f840e2318b821074478b28849181636

图 9-20 作业实例表信息

当前的 Job Instance 的 ID 为 158，作业名字为 externalFlowStepJob。

作业执行器表信息参见图 9-21。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	CREATE_TIME
1	160	2	158	2013-10-14 22:59:36

图 9-21 作业执行器表信息

当前作业执行器的 ID 为 160，对应的 Job Instance 为 158。

作业步执行器表信息参见图 9-22。

	STEP_EXECUTION_ID	STEP_NAME	JOB_EXECUTION_ID	START_TIME	END_TIME
1	248	externalFlowStepJob.flow1	160	2013-10-14 22:59:36	2013-10-14 22:59:37
2	249	decompressStep	160	2013-10-14 22:59:36	2013-10-14 22:59:36
3	250	verifyStep	160	2013-10-14 22:59:36	2013-10-14 22:59:37
4	251	readWriteStep	160	2013-10-14 22:59:37	2013-10-14 22:59:37
5	252	cleanStep	160	2013-10-14 22:59:37	2013-10-14 22:59:37

图 9-22 作业步执行器表信息

此处大家注意，作业步"externalFlowStepJob.flow1"是 flow 中定义的三个 Step 的父，当 flow1 中定义三个作业步执行完毕后，作业步"externalFlowStepJob.flow1"才执行完成，请注意看开始时间和结束时间。

"externalFlowStepJob.flow1"和"flow1"中 step 的生命周期参见图 9-23。

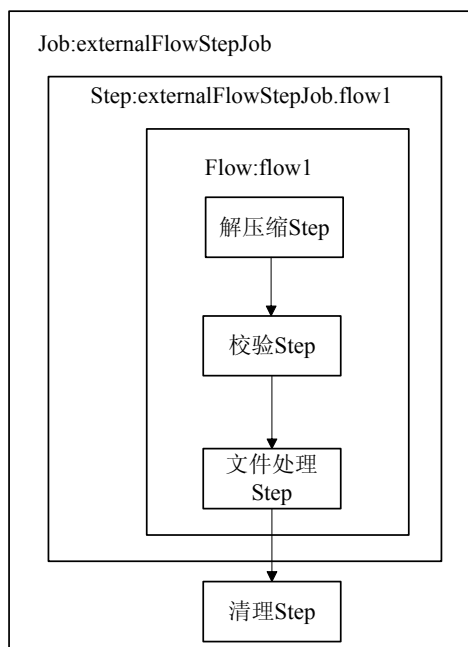


图 9-23 "externalFlowStepJob.flow1"和"flow1"中 step 的生命周期

9.4.3 JobStep

JobStep 是指 Step 元素使用外部定义的 Job，JobStep 和 FlowStep 的区别在于前者将 Job 作为一个单独的 Step 在 Job 执行期间出现（该单独的 Step 将 Job 中定义的所有的 Step 包装在一个大的 Step 中；同时内部的 Job 在运行期间是一个完整的 Job 执行，同时又作为 Step 的子元素出现）；而后者将 Flow 作为一个单独的 Step 在 Job 执行期间出现（该单独的 Step 将 Flow 中定义的所有的 Step 包装在一个大的 Step 中）。

接下来我们给出 JobStep 的示例参见代码清单 9-15。

完整配置参见文件：ch09/job/job-external-job.xml。

代码清单 9-15 JobStep 示例

```

1.     <job id="externalJobJobStep" >
2.         <step id="externalJobJob.flow1" next="cleanStep">
3.             <job ref="baseJob" />
4.         </step>
5.         <step id="cleanStep">
6.             <tasklet ref="cleanTasklet" />
7.         </step>
8.     </job>
9.

```



```

10.     <job id="baseJob">
11.         <step id="decompressStep" parent="abstractDecompressStep"
            next="verifyStep" >
12.             <tasklet ref="decompressTasklet" />
13.         </step>
14.         <step id="verifyStep" next="readWriteStep">
15.             <tasklet ref="verifyTasklet" />
16.         </step>
17.         <step id="readWriteStep" parent="parentReadWriteStep"/>
18.     </job>

```

本示例中定义两个 Job，分别是 baseJob、externalJobJobStep。baseJob 定义了三个作业步，分别是压缩、校验、读/写；在 externalJobJobStep 中定义了两个作业步，externalJobJob.flow1 和清理作业步，前者引用了 baseJob 作业作业步的实现。

因此在 externalJobJobStep 执行时候，会有两个 Job 执行，分别是 baseJob、externalJobJobStep。

执行使用 JobStep 定义的 Job，查看数据库的源信息可以看到，作业 externalJobJobStep 先后执行了 4 个 Step。（执行代码入口：test.com.juxtapose.example.ch09.JobLaunchExternalJobStep）作业实例表信息参见图 9-24。


	 JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
1	161	0	externalJobJobStep	604a64af02993207746dc96cdba7a5c0
2	162	0	baseJob	604a64af02993207746dc96cdba7a5c0

图 9-24 作业实例表信息

当前的 Job Instance 有两个，对应的 ID 分别为 161,162，作业名字为 externalJobJobStep、baseJob。

作业执行器表信息参见图 9-25。

	 JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	CREATE_TIME
1	163	2	161	2013-10-14 23:28:45
2	164	2	162	2013-10-14 23:28:45

图 9-25 作业执行器表信息

当前作业执行器的 ID 为 163，对应的 Job Instance 为 161。

当前作业执行器的 ID 为 164，对应的 Job Instance 为 162。

作业步执行器表信息参见图 9-26。


	 STEP_EXECUTION_ID	STEP_NAME	JOB_EXECUTION_ID	START_TIME	END_TIME
1	258	externalJobJob.flow1	163	2013-10-14 23:28:45	2013-10-14 23:28:46
2	259	decompressStep	164	2013-10-14 23:28:45	2013-10-14 23:28:45
3	260	verifyStep	164	2013-10-14 23:28:46	2013-10-14 23:28:46
4	261	readWriteStep	164	2013-10-14 23:28:46	2013-10-14 23:28:46
5	262	cleanStep	163	2013-10-14 23:28:47	2013-10-14 23:28:47

图 9-26 作业步执行器表信息

此处大家注意，作业步"externalJobJob.flow1"是 baseJob 中定义的三个 Step 的父，当 baseJob 中定义的三个作业步执行完毕后，作业步" externalJobJob.flow1"才执行完成，请注意看开始时间和结束时间。

" externalJobJob.flow1"和"baseJob"中 step 的生命周期参见图 9-27。

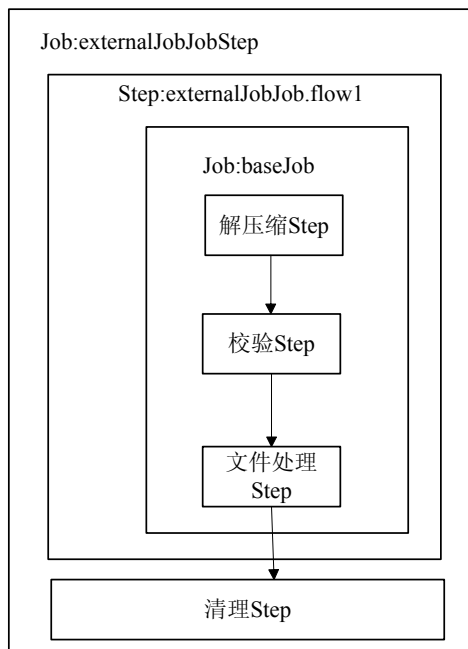


图 9-27 "externalJobJob.flow1"和"baseJob"中 step 的生命周期

9.5 Step 数据共享

Execution Context 是 Spring Batch 框架提供的持久化与控制的 key/value 对，能够让开发者在 Step Execution 或 Job Execution 中保存需要进行持久化的状态。

Execution Context 分为两类：一类是 Job Execution 的上下文（对应表：BATCH_JOB_EXECUTION_CONTEXT）；另一类是 Step Execution 的上下文（对应表：BATCH_STEP_EXECUTION_CONTEXT）。两类上下文之间的关系：一个 Job Execution 对应一个 Job Execution 的上下文；每个 Step Execution 对应一个 Step Execution 上下文；同一个 Job 中的 Step Execution 公用 Job Execution 的上下文。

因此如果同一个 Job 的不同 Step 间需要共享数据，则可以通过 Job Execution 的上下文来共享数据。利用 Execution Context 中的 key/value 对可以重新启动 Job；也可以利用 Execution Context 在不同的作业步 Step 之间进行数据功能。

Spring Batch 中可以通过 tasklet、reader、write、processor、listener 中访问 Execution Context 对象，在不同的 Step 中可以将数据写入 Context 或者从 Context 读取。

Job 上下文、Step 上下文之间的关系图参见图 9-28。

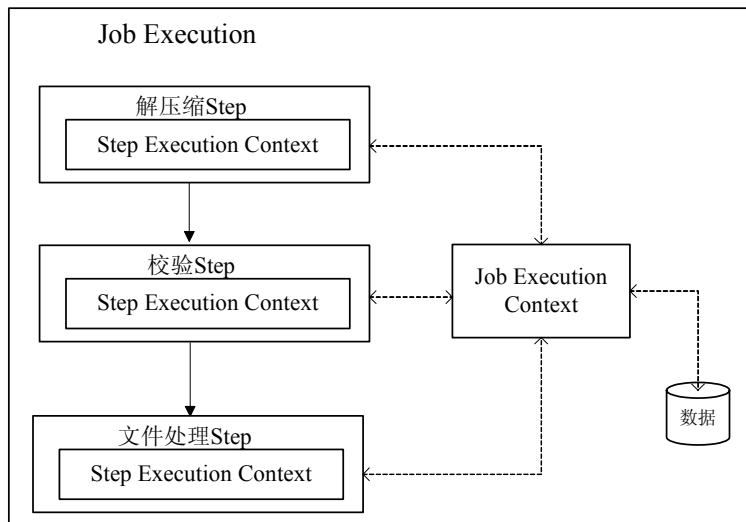


图 9-28 Job 上下文、Step 上下文之间关系

Job Execution Context 在整个 Job 的执行期间存在，不同的 Step 可以将数据存入 Job 上下文中，Job Execution Context 在执行期间会将数据保存到 DB 中，在 Job 重启的时候能够恢复 Job 的状态。

在 Execution Context 中写入、读取数据的典型用法参见代码清单 9-16。

代码清单 9-16 ExecutionContext 中写入、读取数据的典型用法

```
1. executionContext.putString(Constant.READ_FILE, workDirectory + readFileName);
2. String readFile = executionContext.getString(Constant.VERIFY_STATUS);
```

其中，1 行：将数据放入执行上下文中。

2 行：从数据上下文中获取数据。

接下来的示例展示如何在 Tasklet 中将共享数据放入 Job Execution Context 中；然后使用 StepExecutionListener 从 Execution Context 中访问数据。

VerifyTasklet 是校验作业步的实现，参见代码清单 9-17，在该 Step 中将校验的信息存入 Job 上下文。

完整代码参见：com.juxtapose.example.ch09.tasklet.VerifyTasklet。

代码清单 9-17 VerifyTasklet 类实现

```
1. public class VerifyTasklet implements Tasklet {
2.     @Override
```

```

3.     public RepeatStatus execute(StepContribution contribution,
4.         ChunkContext chunkContext) throws Exception {
5.         String status = creditService.verify(outputDirectory, readFileName);
6.         if (null != status) {
7.             chunkContext.getStepContext().getStepExecution()
8.                 .getExecutionContext().put(Constant.VERITY_STATUS, status);
9.         }
10.        return RepeatStatus.FINISHED;
11.    }
12.    .....
13. }

```

其中，7~8 行：将校验的状态存入 Job ExecutionContext 中，方便后面的拦截器能够获取校验的状态，根据校验状态设置 Step 的退出值。

VerifyStepExecutionListener 拦截器从 Job ExecutionContext 中获取校验信息。VerifyStepExecutionListener 的类实现参见代码清单 9-18。

完整代码参见：com.juxtapose.example.ch09.listener.VerifyStepExecutionListener。

代码清单 9-18 VerifyStepExecutionListener 类实现

```

1.  public class VerifyStepExecutionListener implements StepExecutionListener {
2.      @Override
3.      public void beforeStep(StepExecution stepExecution) { }
4.
5.      @Override
6.      public ExitStatus afterStep(StepExecution stepExecution) {
7.          String status =
8.              stepExecution.getExecutionContext().getString(Constant.
9.                  VERITY_STATUS);
10.         if(null != status){
11.             return new ExitStatus(status);
12.         }
13.         return stepExecution.getExitStatus();
14.     }
15. }

```

其中，7~8 行：从 Job ExecutionContext 中获取校验 Step 中的校验结果信息。

9.6 终止 Job

Spring Batch 框架中支持在 Job 的某一个作业步终止作业。截止到目前，前面所有的 Job 均是在 Job 的最后一个 Step 完成 Job 的执行。本节我们展示如何使用 end、stop、fail 元素来根据 ExitStatus 完成 Job 的终止。

end 用来根据 ExitStatus 来正确的完成 Job，使用 end 结束后的 Job 的 BatchStatus 是 COMPLETED，不能重新启动。

stop 用来根据 ExitStatus 来停止 Job，使用 sto 结束后的 Job 的 BatchStatus 是 STOPPED，可以重新启动。

fail 用来根据 ExitStatus 来让 Job 失败，使用 fail 结束后的 Job 的 BatchStatus 是 FAILED，可以重新启动。

终止 end、stop、fail 三者比较参见表 9-8。

表 9-8 终止 end、stop、fail 三者区别

元素	Batch Status	Exit Status	Exit Status 可否更改	可否重启	说 明
end	COMPLETED	COMPLETED	是	否	完成当前的 Step 后，完成当前 Job
stop	STOPPED	STOPPED	否	是	完成当前 Step 后，停止当前 Job
fail	FAILED	FAILED	是	是	完成当前的 Step 后，Job 失败

9.6.1 end

使用 end 元素，可以根据给定的退出状态将 Job 正常的终止掉；通常可以根据业务状态的值将 Job 终止，默认情况下 Job 的退出状态为 COMPLETED，批处理状态为 COMPLETED；可以根据属性 exit-code 来指定 Job 的批处理退出状态值。

元素 end 的 schema 的定义参见图 9-29。

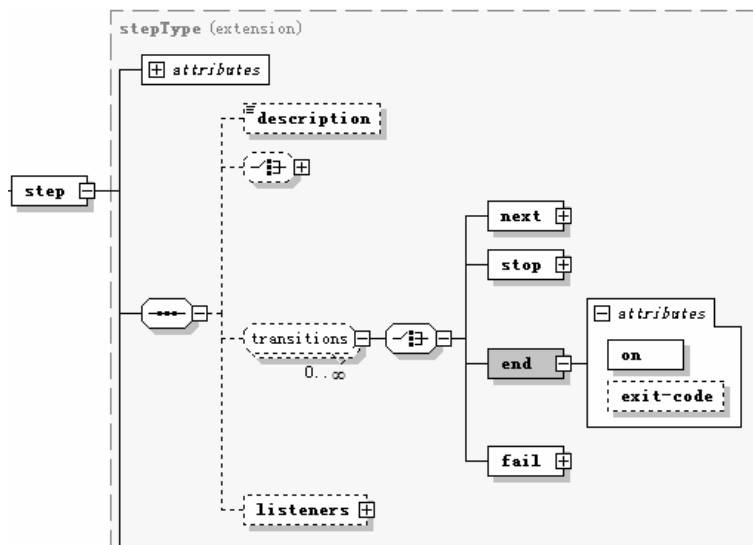


图 9-29 元素 end 的 schema 定义

end 元素的属性说明参见表 9-9。

表 9-9 end 元素的属性说明

属 性	说 明	默 认 值
on	定义作业步的 ExitStatus（退出状态）和 on 属性指定的值匹配的时候，则正常的完成 Job。 on 属性的值可以是任意的字符串，同时支持通配符"*"、"?" "*"：表示退出状态为任何值都满足。 "?"：表示匹配一个字符，如 c?t，当作业的退出状态为 cat 的时候满足，如果是 cabt 则不满足	
exit-code	设置 Job 的退出状态	默认值为： COMPLETED

假设在信用卡账单处理过程中对文件进行校验的作业步进行特殊的处理，如果校验的 ExitStatus 为"FAILED"，则直接使用 end 完成当前的 Job。

代码清单 9-19 展示了如何使用 end 元素终止 Job。

完整配置参见文件：ch09/job/job-conditional-end.xml。

代码清单 9-19 Job 中使用 end 元素

```

1. <job id="conditionalEndJob" >
2.     <step id="decompressStep" parent="abstractDecompressStep"
       next="verifyStep" >
3.         <tasklet ref="decompressTasklet" />
4.     </step>
5.     <step id="verifyStep" >
6.         <tasklet ref="verifyTasklet" />
7.         <end on="FAILED" exit-code="endExitCode"/>
8.         <next on="SKIPTOCLEAN" to="cleanStep" />
9.         <next on="*" to="readWriteStep" />
10.        <listeners>
11.            <listener ref="verifyStepExecutionListener" />
12.        </listeners>
13.    </step>
14.    <step id="readWriteStep" parent="parentReadWriteStep"
       next="cleanStep" />
15.    <step id="cleanStep">
16.        <tasklet ref="cleanTasklet" />
17.    </step>
18. </job>

```

其中，7 行：使用 end 元素终止 Job；属性 on 表示当作业步 verifyStep 的退出状态为“FAILED”的时候将会终止 Job 的执行；属性 exit-code 可以指定 Job 的退出状态。

执行 Job 后，可以查看作业执行器的信息，具体参见图 9-30。Job 的 BatchStatus 为“COMPLETED”，退出状态 EXIT_CODE 为“endExitCode”。

	JOB_EXECUTION_ID	VERSION	STATUS	EXIT_CODE	JOB_INSTANCE_ID
1	168	2	COMPLETED	endExitCode	166

图 9-30 作业执行器的信息

9.6.2 stop

使用 stop 元素，可以根据给定的退出状态将 Job 停止掉；通常可以根据业务状态的值将 Job 停止，默认情况下 Job 的退出状态为 STOPPED，批处理状态也为 STOPPED；可以根据属性 restart 来指定 Job 重启时候从哪个 Step 开始执行。

元素 stop 的 schema 的定义参见图 9-31。

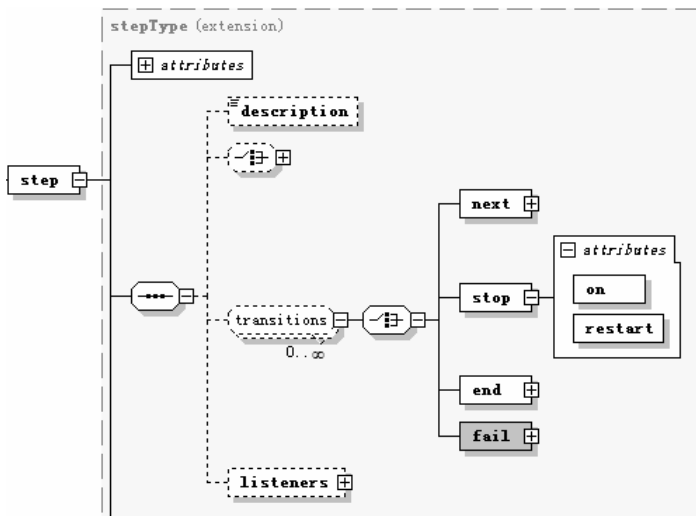


图 9-31 元素 stop 的 schema 定义

stop 元素的属性说明参见表 9-10。

表 9-10 stop 元素的属性说明

属 性	说 明	默 认 值
on	定义作业步的 ExitStatus（退出状态）和 on 属性指定的值匹配的时候，停止当前的 Job。 on 属性的值可以是任意的字符串，同时支持通配符"*"、"?" "*": 表示退出状态为任何值都满足。 "?: 表示匹配一个字符，如 c?t, 当作业的退出状态为 cat 的时候满足，如果是 cabt 则不满足	
restart	指定 Job 重启的时候，从哪个 Step 开始执行	

假设在信用卡账单处理过程中对文件进行校验的作业步进行特殊的处理，如果校验的

ExitStatus 为"FAILED", 则直接使用 stop 停止当前的 Job, 并使用属性 restart 来制定重启时候需要执行的 Step。

代码清单 9-20 展示了如何使用 stop 元素停止 Job。

完整配置参见文件: ch09/job/ job-conditional-stop.xml。

代码清单 9-20 使用 stop 元素停止 Job

```
1. <job id="conditionalStopJob" >
2.     <step id="decompressStep" parent="abstractDecompressStep"
3.         next="verifyStep" >
4.         <tasklet ref="decompressTasklet" />
5.     </step>
6.     <step id="verifyStep" >
7.         <tasklet ref="verifyTasklet" />
8.         <stop on="COMPLETED" restart="readWriteStep"/>
9.         <next on="SKIPTOCLEAN" to="cleanStep" />
10.        <next on="*" to="readWriteStep" />
11.        <listeners>
12.            <listener ref="verifyStepExecutionListener" />
13.        </listeners>
14.    </step>
15.    <step id="readWriteStep" parent="parentReadWriteStep"
16.        next="cleanStep" />
17.    <step id="cleanStep">
18.        <tasklet ref="cleanTasklet" />
19.    </step>
20.</job>
```

其中,7 行:属性 on 定义满足的 ExitStatus 为" COMPLETED "的时候停止 Job;属性 restart 定义 Job 重启时候执行的 Step。

执行 Job 后,可以查看作业执行器的信息,具体参见图 9-32。Job 的 BatchStatus 为“STOPPED”,退出状态 EXIT_CODE 为“STOPPED”。


	 JOB_EXECUTION_ID	VERSION	STATUS	EXIT_CODE	JOB_INSTANCE_ID
1	169	2	STOPPED	STOPPED	167

图 9-32 作业执行器的信息

9.6.3 fail

使用 fail 元素,可以根据给定的退出状态将 Job 正确地终止掉;通常可以根据业务状态的值得 Job 终止,默认情况下 Job 的退出状态为 FAILED,批处理状态为 FAILED;可以根据属性 exit-code 来指定 Job 的批处理退出状态值。

元素 fail 的 schema 的定义参见图 9-33。

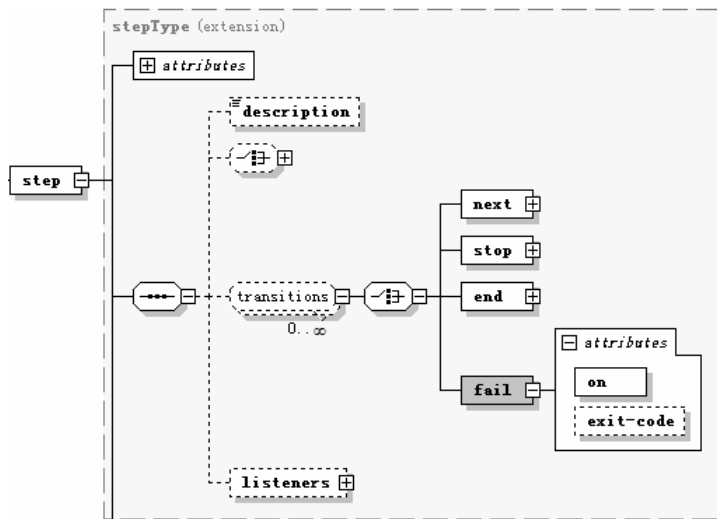


图 9-33 元素 fail 的 schema 的定义

fail 元素的属性说明参见表 9-11。

表 9-11 fail 元素的属性说明

属 性	说 明	默 认 值
on	<p>定义作业步的 ExitStatus（退出状态）和 on 属性指定的值匹配的时候，以失败的方式完成 Job。</p> <p>on 属性的值可以是任意的字符串，同时支持通配符"*"、"?"</p> <p>"*": 表示退出状态为任何值都满足。</p> <p>"?": 表示匹配一个字符，如 c?t, 当作业的退出状态为 cat 的时候满足，如果是 cabt 则不满足</p>	
exit-code	设置 Job 的退出状态	默认值为： FAILED

假设在信用卡账单处理过程中对文件进行校验的作业步进行特殊的处理，如果校验的 ExitStatus 为"FAILED"，则直接使用 fail 终止当前的 Job。

代码清单 9-21 展示了如何使用 fail 元素终止 Job。

完整配置参见文件：ch09/job/job-conditional-fail.xml。

代码清单 9-21 使用 fail 元素终止 Job

```

1. <job id="conditionalFailJob" >
2.     <step id="decompressStep" parent="abstractDecompressStep"
       next="verifyStep" >
3.         <tasklet ref="decompressTasklet" />
4.     </step>

```

```

5.      <step id="verifyStep" >
6.          <tasklet ref="verifyTasklet" />
7.          <fail on="FAILED" exit-code="EARLY TERMINATION"/>
8.          <next on="SKIPTOCLEAN" to="cleanStep" />
9.          <next on="*" to="readWriteStep" />
10.         <listeners>
11.             <listener ref="verifyStepExecutionListener" />
12.         </listeners>
13.     </step>
14.     <step id="readWriteStep" parent="parentReadWriteStep"
15.         next="cleanStep" />
16.     <step id="cleanStep">
17.         <tasklet ref="cleanTasklet" />
18.     </step>
19. </job>

```

其中，7 行：属性 on 定义满足的 ExitStatus 为 " FAILED "的时候停止 Job；属性 exit-code 定义 Job 终止时候的退出状态为 “EARLY TERMINATION”。

执行 Job 后，可以查看作业执行器的信息，具体参见图 9-34。Job 的 BatchStatus 为 “FAILED”，退出状态 EXIT_CODE 为 “EARLY TERMINATION”。


		JOB_EXECUTION_ID	VERSION	STATUS	EXIT_CODE	JOB_INSTANCE_ID
1		174	2	FAILED	EARLY TERMINATION	172

图 9-34 作业执行器的信息

健壮 Job

批处理要求 Job 必须有较强的健壮性，通常 Job 是批量处理数据、无人值守的，这要求在 Job 执行期间能够应对各种发生的异常、错误，并对 Job 执行进行有效的跟踪。一个健壮的 Job 通常需要具备如下的几个特性。

(1) 容错性

在 Job 执行期间非致命的异常，Job 执行框架应能够进行有效的容错处理，而不是让整个 Job 执行失败；通常只有致命的、导致业务不正确的异常才可以终止 Job 的执行。

(2) 可追踪性

Job 执行期间任何发生错误的地方都需要进行有效的记录，方便后期对错误点进行有效的处理。例如在 Job 执行期间任何被忽略处理的记录行需要被有效地记录下来，应用程序维护人员可以针对被忽略的记录后续做有效的处理。

(3) 可重启性

Job 执行期间如果因为异常导致失败，应该能够在失败的点重新启动 Job；而不是从头开始重新执行 Job。

Spring Batch 框架提供了支持上面所有能力的特性，包括 Skip（跳过记录处理）、Retry（重试给定的操作）、Restart（从错误点开始重新启动失败的 Job），具体描述参见表 10-1。

表 10-1 健壮 Job 特性

特 性	功 能	适用时机	适用场景
Skip	跳过错误的记录行，保证 Job 能够正确地执行	适用于非致命的异常	面向 Chunk 的 Step
Retry	重试给定的操作，比如短暂的网络异常、并发异常等	适用于短暂的异常，经过重试之后该异常可能会不再重现	面向 Chunk 的 Step 或者应用代码
Restart	Job 执行失败后，重新启动 Job 实例	因异常、错误导致 Job 失败后	Job 执行重新启动

Skip，在对 Flat 类型的文件处理期间，如果文件中某行的格式不能满足要求，可以通过 Skip 跳过该行记录的处理，让 Processor 能够顺利地处理其余的记录行。

Retry，将给定的操作进行多次重试，在某些情况下操作因为短暂的异常导致执行失败，如网络连接异常、并发处理异常等，可以通过重试的方式避免单次的失败，下次执行操作的时候网络恢复正常，不再有并发的异常，这样通过重试的能力可以有效地避免这类短暂的异常。

Restart，在 Job 执行失败后，可以通过重启功能来继续完成 Job 的执行。在重启时候，

批处理框架允许在上次执行失败的点重新启动 Job，而不是从头开始执行，这样可以大幅提高 Job 执行的效率。

10.1 跳过 Skip

Step 执行期间 read、process、write 发生的任何异常都会导致 Step 执行失败，进而导致作业的失败。批处理作业的自动化、定时触发，有特定的执行时间窗口特性，决定了尽可能地减少 Job 的失败。设想信用卡对账单的处理的业务场景，银行每天需要处理海量的对账文件，如果对账文件中有少量的一行或者几行错误格式的记录，在真正进行作业处理的时候，不希望因为几行错误的记录而导致整个作业的失败；而是希望将这几行没有处理的记录跳过去，让整个 Job 正确执行，对于错误的记录则通过日志的方式记录下来后续进行单独的处理。

Spring Batch 框架通过属性 skip-limit、skippable-exception-classes、skip-policy 来完成异常跳过的能力，具体属性参见表 10-2。

表 10-2 Skip 属性描述

属性/元素	功能说明
skippable-exception-classes	定义允许跳过的异常，碰到该类型异常时候，不会导致 Job 失败；而是跳过当前记录的处理，保证 Job 继续正确的执行
include	skippable-exception-classes 的子元素，用以表示包括在内的异常
exclude	skippable-exception-classes 的子元素，用以表示排出在内的异常，通常用来定义某一类型的子异常
skip-limit	跳过限制次数，当超过该次数后再发生异常会导致 Job 失败
skip-policy	Job 的跳过策略，根据该策略判断是否允许跳过异常

skippable-exception-classes：定义记录跳过的异常，可以定义一组异常，如果发生了定义的异常或者子类异常都不会导致作业失败。

skip-limit：任务处理发生异常时，允许跳过的最大次数。

skip-policy：当默认的按照次数跳过策略不能满足需求时，可以配置自定义跳过策略，需要实现接口：`org.springframework.batch.core.step.skip.SkipPolicy`。

10.1.1 配置 Skip

何种类型的异常能够跳过处理，通常由开发人员根据具体的业务场景确定，比如单条记录不完整，或者完整但是格式不正确导致无法正确解析。可以使用属性 `skippable-exception-classes` 来定义当发生那些类型的异常时，避免 Job 失败。

代码清单 10-1 给出了对账单格式错误文件示例。

代码清单 10-1 错误格式对账单示例

```
1. 4047390012345678,tom,100.00,xxx-yyy-zzz,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.20,2013-2-9 15:15:37
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road
```

其中，1 行：交易日期格式有误。

4 行：记录不完整，缺少交易的地址列信息。

如果记录行数错误较多，即使成功执行完毕 Job，没有任何意义，因为错误记录数太多导致后续的手工修复可能更复杂。此时最好的处理方式是让 Job 执行失败，然后修复记录文件重新执行 Job 作业。Spring 框架提供了最大跳过记录数的属性 skip-limit，当跳过的记录数大于设定的值后，Job 作业将会失败。

配置 Skip

配置 Skip 的示例参见代码清单 10-2。设置允许跳过的最大记录为 20 条，在发生 java.lang.RuntimeException 但不包括 java.io.FileNotFoundException 的时候跳过处理。

完整配置参见文件：/ch10/job/job-step-skip.xml。

代码清单 10-2 配置 Skip

```
1. <job id="skipJob">
2.     <step id="skipStep">
3.         <tasklet>
4.             <chunk reader="reader" processor="processor" writer="writer"
5.                 commit-interval="1" skip-limit="20">
6.                 <skippable-exception-classes>
7.                     <include class="java.lang.RuntimeException" />
8.                     <exclude class="java.io.FileNotFoundException" />
9.                 </skippable-exception-classes>
10.            </chunk>
11.        </tasklet>
12.    </step>
13. </job>
14.
15. <bean:bean id="processor"
16.     class="com.juxtapose.example.ch10.skip.
    RadomExceptionItemProcessor" />
```

其中，5 行：异常发生时，允许跳过最大的异常次数为 20 次，如果超过 20 次后，再次发生的异常会导致 Job 的失败。

6~9 行：include 定义支持跳过的异常，exclude 定义排出在外的异常；当发生了任何 RuntimeException 类型异常（同时 FileNotFoundException 排出在外）时会跳过处理，但是当

已经跳过 20 次后，第 21 次发生异常的时候，会导致 Job 的失败，因为超过了 skip-limit="20" 的限制。

15~16 行：自定义一个 ItemProcessor 实现类 RadomExceptionItemProcessor，该类会随机发生 RuntimeException 类型异常。

com.juxtapose.example.ch10.skip.RadomExceptionItemProcessor 实现代码参见代码清单 10-3。

代码清单 10-3 RadomExceptionItemProcessor 类实现

```
1. public class RadomExceptionItemProcessor implements
   ItemProcessor<String,String> {
2.     Random ra = new Random();
3.
4.     public String process(String item) throws Exception {
5.         int i = ra.nextInt(10);
6.         System.out.println("Process " + item + "; Random i=" + i);
7.         if(i%2 == 0){
8.             throw new RuntimeException("make error!");
9.         }else{
10.            return item;
11.        }
12.    }
13. }
```

其中，5 行：定义随机数。

7~10 行：根据对 2 取模，抛出异常或者正确返回记录。

说明：发生的异常可能来自 Read、Process、Write 执行阶段，即 Read、Process、Write 任何操作发生的异常都可能导致 Skip 发生。

10.1.2 跳过策略 SkipPolicy

根据 skip-limit、skippable-exception-classes 可以配置简单的跳过逻辑处理，通常情况下可能需要复杂的跳过逻辑配置，Spring Batch 框架提供了友好的扩展策略，通过实现接口 SkipPolicy 可以自定义跳过策略。

skip-limit 设置限制跳过次数，默认使用的跳过策略为 LimitCheckingItemSkipPolicy。

接口 org.springframework.batch.core.step.skip.SkipPolicy 定义参见代码清单 10-4。

代码清单 10-4 SkipPolicy 接口定义

```
1. public interface SkipPolicy {
2.     boolean shouldSkip(Throwable t, int skipCount) throws SkipLimitExceeded
       Exception;
3. }
```

其中，2 行：接口核心方法，根据给定的异常和已经跳过的记录数，判断当前发生的异

常是否被跳过处理；t 表示发生的异常，skipCount 表示已经跳过的记录数。

系统提供的默认 SkipPolicy 实现列表参见表 10-3。

表 10-3 SkipPolicy 默认实现

SkipPolicy 默认实现	功能说明
AlwaysSkipItemSkipPolicy	发生任何异常都会导致跳过记录处理 org.springframework.batch.core.step.skip.AlwaysSkipItemSkipPolicy
CompositeSkipPolicy	组合跳过策略，可以将多个跳过策略组合在一起使用，按照顺序判断是否应该跳过该记录；多个组合策略中只要有一个允许跳过，则组合策略允许跳过该记录的处理 org.springframework.batch.core.step.skip.CompositeSkipPolicy
ExceptionClassifierSkipPolicy	为不同的异常定义不同的跳过策略 org.springframework.batch.core.step.skip.ExceptionClassifierSkipPolicy
LimitCheckingItemSkipPolicy	根据设置的次数决定异常是否能被跳过处理 org.springframework.batch.core.step.skip.LimitCheckingItemSkipPolicy
NeverSkipItemSkipPolicy	发生任何异常都不会被跳过 org.springframework.batch.core.step.skip.NeverSkipItemSkipPolicy

接下来，以 AlwaysSkipItemSkipPolicy 为例展示如何使用 SkipPolicy，详细参见代码清单 10-5。

完整配置文件参见：/ch10/job/job-step-skip.xml。

代码清单 10-5 配置 SkipPolicy

```
1.     <job id="skipPolicyJob">
2.         <step id="skipPolicyStep">
3.             <tasklet>
4.                 <chunk reader="reader" processor="processor" writer="writer"
5.                     commit-interval="2" skip-policy="alwaysSkipPolicy">
6.                     </chunk>
7.             </tasklet>
8.         </step>
9.     </job>
10.
11.     <bean:bean id="alwaysSkipPolicy"
12.         class="org.springframework.batch.core.step.skip.
            AlwaysSkipItemSkipPolicy"/>
```

其中，5 行：使用 alwaysSkipPolicy 对象作为跳过处理策略。

11~12：定义 alwaysSkipPolicy，默认实现为 AlwaysSkipItemSkipPolicy，表示发生的任何异常都不会中断 Step、Job 的处理。

执行 skipPolicyJob 作业期间，任何发生的异常都会被忽略掉，保证 Job 的正确执行，但

是对于跳过的记录需要通过日志或者其他的方式记录下来，方便后期的对账处理或者人工干预。在 10.1.3 章节我们将展示如果通过拦截器记录跳过的处理。

10.1.3 跳过拦截器

`SkipListener` 在 `Chunk` 处理阶段抛出跳过定义的异常时候触发，在 `Chunk` 的读、处理、写阶段发生的异常都会触发该拦截器。接口 `SkipListener` 定义参见代码清单 10-6。

代码清单 10-6 `SkipListener` 接口定义

```
1. public interface SkipListener<T,S> extends StepListener {
2.     void onSkipInRead(Throwable t);
3.     void onSkipInWrite(S item, Throwable t);
4.     void onSkipInProcess(T item, Throwable t);
5. }
```

`SkipListener` 操作说明与 `Annotation` 定义参见表 10-4。

表 10-4 `SkipListener` 操作说明与 `Annotation`

操 作	操作说明	Annotation
<code>onSkipInRead(Throwable t)</code>	在读阶段发生异常并且配置了异常可以跳过时候触发该操作	@ OnSkipInRead
<code>onSkipInWrite(S item, Throwable t)</code>	在写阶段发生异常并且配置了异常可以跳过时候触发该操作	@ OnSkipInWrite
<code>onSkipInProcess(T item, Throwable t)</code>	在处理阶段发生异常并且配置了异常可以跳过时候触发该操作	@ OnSkipInProcess

`SkipListener` 系统实现参见表 10-5。

表 10-5 `SkipListener` 默认实现

<code>SkipListener</code> 默认实现	功能说明
<code>CompositeSkipListener</code>	组合跳过拦截器实现，可以定义一组跳过拦截器，依照顺序执行 <code>org.springframework.batch.core.listener.CompositeSkipListener<T, S></code>
<code>MulticasterBatchListener</code>	同时实现 <code>StepExecutionListener</code> , <code>ChunkListener</code> , <code>ItemReadListener<T></code> , <code>ItemProcessListener<T, S></code> , <code>ItemWriteListener<S></code> , <code>SkipListener<T, S></code> 六个接口组合策略的拦截器 <code>org.springframework.batch.core.listener.MulticasterBatchListener<T, S></code>
<code>SkipListenerSupport</code>	跳过拦截器的默认执行，可以继承该类仅实现关心的方法 <code>org.springframework.batch.core.listener.SkipListenerSupport<T, S></code>

跳过拦截器执行时机

`Skip` 拦截器并非在读、处理、写阶段发生异常后立刻执行；而是在批操作事务提交正确

之前才执行。下面我们分析一下 Spring Batch 框架如此实现的原因：Spring Batch 框架是面向批的操作，每一批的处理被隐性地封装在同一个事务中，当发生跳过的异常时候，在批后面的操作过程中可能出现其他类型的错误导致发生回滚，即作业失败，在这种场景下我们不希望 Skip 拦截器在前面已经执行；如果 Skip 拦截器已经执行了导致的结果是，整个作业失败但有部分跳过的记录做了 Skip 拦截器的动作，导致状态不一致。因此 Skip 拦截器在 Chunk 能正确提交事务之前被触发，而不是读、处理、写阶段发生异常的时候就触发。

说明：跳过记录的处理，并不仅仅在读阶段发生跳过异常，在处理、写阶段发生跳过异常时候同样会触发 Skip 拦截器。

Read 阶段发生跳过异常：此时 Spring Batch 框架继续调用 read 操作获取下一个条目。

Process 阶段发生跳过异常：Spring Batch 框架进行回滚当前批操作，并重新提交读到的数据（发生异常的那条数据除外）。

Write 阶段发生跳过异常：因为写数据是批量提交的，不知道哪条记录发生了跳过异常，Spring Batch 框架将为每一条数据启用一个单独的事务进行数据的重新处理。写阶段发生跳过异常的处理参见图 10-1。

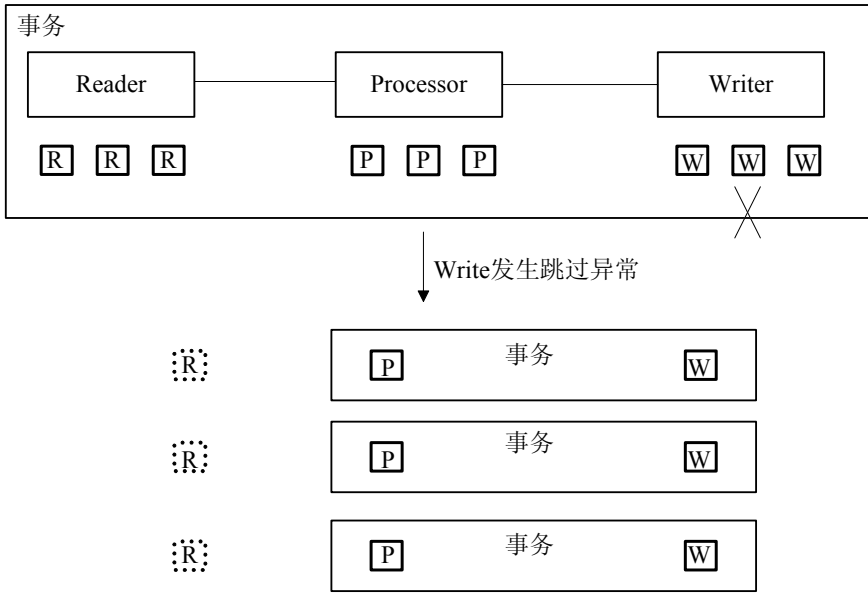


图 10-1 写阶段发生跳过异常的处理

在实际批处理作业中，对于跳过的记录需要记录下来，在 Job 完成后需要进行后续的跟踪处理或者人工干预。在信用卡对帐单作业处理工程中需要将跳过的记录存放在数据库中。

实现 SkipListener 记录跳过的记录存放在数据库。

com.juxtapose.example.ch10.skip.DBSkipListener 类定义参见代码清单 10-7。

代码清单 10-7 DBSkipListener 类实现

```
1. public class DBSkipListener implements SkipListener<String, String> {
2.     private JdbcTemplate jdbcTemplate;
3.
4.     public void onSkipInRead(Throwable t) {
5.         if (t instanceof FlatFileParseException) {
6.             jdbcTemplate.update("insert into skipbills "
7.                 + "(line,content) values (?,?)",
8.                 ((FlatFileParseException) t).getLineNumber(),
9.                 ((FlatFileParseException) t).getInput());
10.        }
11.    }
12.
13.    public void onSkipInWrite(String item, Throwable t) {}
14.    public void onSkipInProcess(String item, Throwable t) {}
15.    .....
16. }
```

其中，6~10 行：将读记录跳过的行信息记录数据库。

13~14 行：对于写和处理阶段的跳过操作忽略处理。

跳过拦截器声明配置参见代码清单 10-8。

完整配置文件参见：/ch10/job/job-step-skip-listener.xml。

代码清单 10-8 配置跳过拦截器

```
1. <job id="dbSkipJob">
2.     <step id="dbSkipStep">
3.         <tasklet>
4.             <chunk reader="radomExceptionAutoReader" processor="processor"
5.                 writer="writer" commit-interval="5" skip-limit="20">
6.                 <skippable-exception-classes>
7.                     <include class="org.springframework.batch.item.file.
8.                         FlatFileParseException" />
9.                 </skippable-exception-classes>
10.                <listeners>
11.                    <listener ref="dbSkipListener"></listener>
12.                </listeners>
13.            </chunk>
14.        </tasklet>
15.    </step>
16. </job>
17. <bean:bean id="radomExceptionAutoReader"
18.     class="com.juxtapose.example.ch10.
19.         RadomExceptionAutoReader" />
20. <bean:bean id="dbSkipListener"
```

```

20.         class="com.juxtapose.example.ch10.step.listener.
           DBSkipListener" >
21.         <bean:property name="jdbcTemplate" ref="jdbcTemplate">
           </bean:property>
22.     </bean:bean>

```

其中，7~8 行：定义允许跳过的异常为 `org.springframework.batch.item.file.FlatFileParseException`。

11 行：定义跳过拦截器为 `dbSkipListener`，负责将跳过的行记录到数据库中。

17~18 行：定义读 `radomExceptionAutoReader`，随机产生类型 `FlatFileParseException` 的异常抛出。

19~22 行：声明跳过拦截器，使用 Spring 中的 `jdbcTemplate` 负责将记录持久化到数据库中。

持久化信息配置声明参见代码清单 10-9。

完整配置文件参见：`/ch10/job-context-db.xml`。

代码清单 10-9 配置持久化信息

```

1.     <jdbc:initialize-database data-source="dataSource">
2.         <jdbc:script location="classpath:ch10/db/create-table-skipbills.
           sql" />
3.     </jdbc:initialize-database>
4.
5.     <bean:bean id="jdbcTemplate" class="org.springframework.jdbc.core.
           JdbcTemplate">
6.         <bean:constructor-arg ref="dataSource" />
7.     </bean:bean>

```

其中，3 行：初始化数据库文件 `create-table-skipbills.sql`，负责将拦截器用到的表 `skipbills` 初始化。

5~7 行：定义 Spring 的 `jdbc` 模板，提供执行 `sql` 语句的基本功能。

数据库表定义具体参见文件：`/ch10/db/create-table-skipbills.sql`。

10.2 重试 Retry

Step 执行期间 `read`、`process`、`write` 发生的任何异常都会导致 Step 执行失败，进而导致作业的失败。批处理作业的自动化、定时触发，有特定的执行时间窗口特性，决定了尽可能地减少 Job 的失败。处理任务阶段发生的异常可以让业务失败，也可以通过 `Skip` 的设置，跳过部分异常；但是另外还有部分异常，例如并发对数据库的操作导致的数据库锁的异常（`DeadlockLoserDataAccessException`）、网络不稳定导致的网络连接异常（`java.net.ConnectException`）。这类异常的出现可能在下次重新操作的时候消失，数据库锁的异常在下次操作可能正确恢复，网络不能连接的异常可能在重试几次后恢复正常。因此，这些异常出现的时候，不期望作业

发生异常，而是希望通过几次重试操作，尽可能让 Job 成功执行。

Spring Batch 框架提供了任务重试功能、重试次数限制功能、自定义重试策略以及重试拦截器能力。分别通过属性 `retryable-exception-classes`、`retry-limit`、`retry-policy`、`cache-capacity`、`retry-listeners` 来实现。具体属性描述参见表 10-6。

表 10-6 Retry 属性描述

属性/元素	功能说明
<code>retryable-exception-classes</code>	定义允许重试的异常，碰到该类型异常时候，不会导致 Job 失败；而是重试当前的操作，保证 Job 继续正确地执行
<code>include</code>	<code>retryable -exception-classes</code> 的子元素，用以表示包括在内的异常
<code>exclude</code>	<code>retryable -exception-classes</code> 的子元素，用以表示排出在内的异常，通常用来定义某一类型的子异常
<code>retry-limit</code>	任务最大重试次数，当超过该次数后再发生异常会导致 Job 失败
<code>retry-policy</code>	Job 的重试策略，根据该策略判断是否允许重试该次失败的操作
<code>cache-capacity</code>	存放 <code>RetryContext</code> 的缓存大小，当超过该值时，会发生异常
<code>retry-listeners</code>	定义重试拦截器

`retryable-exception-classes`：定义可以重试的异常，可以定义一组异常，如果发生了定义的异常或者子类异常都会导致重试。

`retry-limit`：任务执行重试的最大次数。

`skip-policy`：定义自定义的重试策略，需要实现接口：`org.springframework.batch.retry.RetryPolicy`。

`cache-capacity`：`retry-policy` 缓存的大小，缓存用于存放重试上下文 `RetryContext`，如果超过配置最大值，会发生异常：`org.springframework.batch.retry.policy.RetryCacheCapacityExceededException`。

`retry-listeners`：配置重试监听器，监听器需要实现接口：`org.springframework.batch.retry.RetryListener`。

10.2.1 配置 Retry

为了展示重试功能，在 Job 的处理阶段模拟发生异常，为了避免无限次地重试该操作，设置限制重试次数为 3 次。

配置 `Retry` 的示例参见代码清单 10-10。

完整配置文件参见：`/ch10/job/job-step-retry.xml`。

代码清单 10-10 配置 `Retry`

```
1.      <job id="retryJob">
2.          <step id="retryStep">
```

```

3.         <tasklet>
4.             <chunk reader="reader" processor="alwaysExceptionItemProcessor"
5.                 writer="writer" commit-interval="1" retry-limit="3">
6.                 <retry-listeners>
7.                     <listener ref="sysoutRetryListener"></listener>
8.                 </retry-listeners>
9.                 <retryable-exception-classes>
10.                    <include class="java.lang.RuntimeException" />
11.                    <exclude class="java.io.FileNotFoundException" />
12.                </retryable-exception-classes>
13.            </chunk>
14.        </tasklet>
15.    </step>
16.</job>
17.    <bean:bean id="sysoutRetryListener"
18.        class="com.juxtapose.example.ch10.retry.SystemOutRetryListener"/>
19.    <bean:bean id="alwaysExceptionItemProcessor"
20.        class="com.juxtapose.example.ch10.retry.
        AlwaysExceptionItemProcessor" />

```

其中，5 行：属性 `retry-limit` 定义最大重试次数为 3 次，当重试超过 3 次后发生的异常会导致 Step 失败。

6~8 行：定义重试操作的拦截器，在 `chunk` 发生重试操作时候，会触发拦截器 `sysoutRetryListener` 操作。

9~12 行：属性 `retryable-exception-classes` 定义重试的异常，可以定义多个异常，`include` 表示能够触发重试的异常，`exclude` 表示不会触发重试的异常。

17~18 行：定义重试拦截器 `sysoutRetryListener`，拦截器仅把信息打印在控制台上。

19~20 行：定义自定义处理器 `alwaysExceptionItemProcessor`，每次操作均会发生异常，用于模拟 `chunk` 发生异常。

`com.juxtapose.example.ch10.retry.AlwaysExceptionItemProcessor` 实现代码参见代码清单 10-11。

代码清单 10-11 AlwaysExceptionItemProcessor 类实现

```

1. public class AlwaysExceptionItemProcessor implements ItemProcessor
   <String,String> {
2.     Random ra = new Random();
3.     public String process(String item) throws Exception {
4.         int i = ra.nextInt(10);
5.         if(i%2 == 0){
6.             System.out.println("Process " + item + "; Random i=" + i + "; ...");
7.             throw new MockARuntimeException("make error!");
8.         }else{
9.             System.out.println("Process " + item + "; Random i=" + i + "; ...");

```

```
10.         throw new MockBRuntimeException("make error!");
11.     }
12. }
13. }
```

其中，4 行：定义随机数。
5~10 行：根据对 2 取模，抛出不同类型的异常。

10.2.2 重试策略 RetryPolicy

根据 `retryable-exception-classes`、`retry-limit` 可以配置简单的重试逻辑处理，通常情况下可能需要复杂的重试逻辑配置，Spring Batch 框架提供了友好的扩展策略，通过实现接口 `RetryPolicy` 可以自定义重试策略。

接口 `org.springframework.batch.retry.RetryPolicy` 定义参见代码清单 10-12。

代码清单 10-12 `RetryPolicy` 接口定义

```
1. public interface RetryPolicy {
2.     boolean canRetry(RetryContext context);
3.     RetryContext open(RetryContext parent);
4.     void close(RetryContext context);
5.     void registerThrowable(RetryContext context, Throwable throwable);
6. }
```

其中，`canRetry()` 判断是否应该重试，`open()` 在重试开始时执行，`close()` 在重试结束时执行，`registerThrowable()` 操作注册异常和重试上下文。

2 行：接口核心方法 `canRetry()`，根据给定的异常和已经跳过的记录数，判断当前发生的异常是否被跳过处理；`t` 表示发生的异常，`skipCount` 表示已经跳过的记录数。

系统提供的默认 `RetryPolicy` 实现列表参见表 10-7。

表 10-7 `RetryPolicy` 系统默认实现

RetryPolicy 默认实现	功能说明
AlwaysRetryPolicy	发生任何异常都会导致重试操作 <code>org.springframework.batch.retry.policy.AlwaysRetryPolicy</code>
NeverRetryPolicy	发生任何异常都会导致重试操作 <code>org.springframework.batch.retry.policy.NeverRetryPolicy</code>
CompositeRetryPolicy	组合重试策略，可以将多个重试策略组合在一起使用，按照顺序判断是否应该重试操作；多个组合策略中只要有一个不允许重试，则组合策略允许重试该操作 <code>org.springframework.batch.retry.policy.CompositeRetryPolicy</code>
ExceptionClassifierRetryPolicy	为不同的异常定义不同的重试策略 <code>org.springframework.batch.retry.policy.ExceptionClassifierRetryPolicy</code>

续表

RetryPolicy 默认实现	功能说明
SimpleRetryPolicy	根据设置的次数决定是否进行重试 org.springframework.batch.retry.policy.SimpleRetryPolicy
TimeoutRetryPolicy	在给定的时间内可以进行重试，超过给定的时间将不会进行重试操作 org.springframework.batch.retry.policy.TimeoutRetryPolicy

接下来以 `ExceptionClassifierRetryPolicy`（为不同的异常定义不同的重试策略）为例展示如何使用 `SkipPolicy`，具体参见代码清单 10-13。

完整配置文件参见：`/ch10/job/job-step-retry.xml`。

代码清单 10-13 配置 `retryPolicy`

```

1.     <job id="retryPolicyJob">
2.         <step id="retryPolicyStep">
3.             <tasklet>
4.                 <chunk reader="reader"
5.                     processor="alwaysExceptionItemProcessor"
6.                     writer="writer" commit-interval="1"
7.                     retry-policy="exceptionClassifierRetryPolicy">
8.             </tasklet>
9.         </step>
10.    </job>

```

其中，6 行：属性 `retry-policy` 定义使用的跳过策略 `exceptionClassifierRetryPolicy`，该策略根据不同的异常可以配置不同的重试次数，具体参见代码清单 10-14。

配置重试策略 `exceptionClassifierRetryPolicy`。

代码清单 10-14 配置重试策略 `exceptionClassifierRetryPolicy`

```

1. <bean:bean id="exceptionClassifierRetryPolicy"
2.     class="org.springframework.retry.policy.
3.     ExceptionClassifierRetryPolicy">
4.     <bean:property name="policyMap">
5.         <bean:map>
6.             <bean:entry key="com.juxtapose.example.ch10.retry.
7.                 MockARuntimeException">
8.                 <bean:bean class="org.springframework.retry.policy.
9.                     SimpleRetryPolicy">
10.                     <bean:property name="maxAttempts" value="3" />
11.                 </bean:bean>
12.             </bean:entry>
13.             <bean:entry key="com.juxtapose.example.ch10.retry.
14.                 MockBRuntimeException">

```

```
12.         <bean:bean class=" org.springframework.retry.policy.  
13.                                     SimpleRetryPolicy">  
14.             <bean:property name="maxAttempts" value="5" />  
15.         </bean:bean>  
16.     </bean:entry>  
17. </bean:map>  
18. </bean:property>  
19. </bean:bean>
```

其中，5~10 行：定义异常 `MockARuntimeException` 的重试策略为 `SimpleRetryPolicy`，最大重试次数为 3 次。

11~16：定义异常 `MockBRuntimeException` 的重试策略为 `SimpleRetryPolicy`，最大重试次数为 5 次。

通过对 `Chunk` 的重试支持，在发生瞬态异常情况下通过重试操作保证 `Job` 执行的稳定性，尽可能避免 `Job` 作业失败的可能。如果 `Step` 的实现不是 `Chunk` 类型操作，而是自定义的 `Tasklet` 操作，可以使用 `Spring Batch` 框架提供的重试模板 `RetryTemplate` 实现重试功能，`RetryTemplate` 的具体使用方式 10.2.4 节会详细介绍。

10.2.3 重试拦截器

`RetryListener` 在 `Chunk` 处理阶段抛出重试定义的异常时候触发，通过拦截器方便在重试动作发生时候进行日志记录、收集重试信息等。可以直接实现接口 `RetryListener`（参见代码清单 10-15），也可以直接继承 `RetryListenerSupport`，通常只需要实现 `onError` 操作，在重试发生错误时触发该操作。

代码清单 10-15 `RetryListener` 接口定义

```
1. public interface RetryListener {  
2.     <T> boolean open(RetryContext context, RetryCallback<T> callback);  
3.     <T> void close(RetryContext context, RetryCallback<T> callback,  
4.         Throwable throwable);  
5.     <T> void onError(RetryContext context, RetryCallback<T> callback,  
6.         Throwable throwable);  
7. }
```

`RetryListener` 核心操作说明参见表 10-8。

表 10-8 `RetryListener` 操作说明

操 作	操作说明	Annotation
<code>open(RetryContext context, RetryCallback<T> callback)</code>	在进入 <code>retry</code> 之前执行该操作，可以在该操作中准备重试需要的资源；如果该操作返回 <code>false</code> ，将会终止本次重试操作，且会抛出异常： <code>org.springframework.batch.retry.TerminatedRetryException</code>	无

续表

操 作	操作说明	Annotation
close(RetryContext context, RetryCallback<T> callback, Throwable throwable)	在 retry 结束之前执行该操作，可以在该方法中关闭在 open 操作中打开的资源	无
onError(RetryContext context, RetryCallback<T> callback, Throwable throwable)	重试发生错误时触发该操作	无

RetryListener 系统实现参见表 10-9。

表 10-9 RetryListener 系统默认实现

RetryListener 默认实现	功能说明
RetryListenerSupport	重试拦截器的默认执行，可以继承该类仅实现关心的方法 org.springframework.batch.retry.listener.RetryListenerSupport

可以使用系统提供的重试拦截器的支持类，简化重试拦截器的实现，仅需要实现业务关心的操作；Spring 整个框架提供了丰富的模板类和支持类来简化业务开发能力。

说明：RetryListener 的 open、close 操作在 Chunk 的 process、write 中即使不发生重试异常也会执行。因为 Chunk 中的 process 和 write 操作默认使用 org.springframework.batch.retry.support.RetryTemplate 进行执行；所以会导致 open、close 操作会每次都执行。

接下来我们实现自定义的重试拦截器用于记录日志信息，并配置重试拦截器，具体参见代码清单 10-16。

完整配置文件参见：/ch10/job/job-step-retry-listener.xml。

代码清单 10-16 配置重试拦截器

```

1.     <job id="retryListenerJob">
2.         <step id="retryStep">
3.             <tasklet>
4.                 <chunk reader="reader" processor="alwaysExceptionItemProcessor"
5.                     writer="writer" commit-interval="1" retry-limit="3">
6.                     <retry-listeners>
7.                         <listener ref="sysoutRetryListener"></listener>
8.                     </retry-listeners>
9.                     .....
10.                </chunk>
11.            </tasklet>
12.        </step>
13.    </job>
14.    <bean:bean id="sysoutRetryListener"

```

```
15.         class="com.juxtapose.example.ch10.retry.SystemOutRetryListener"/>
```

其中，6~8行：定义重试拦截器，在重试操作发生时候会触发该拦截器的操作。

SystemOutRetryListener 的实现代码参见代码清单 10-17。

完整代码参见：com.juxtapose.example.ch10.retry.SystemOutRetryListener。

代码清单 10-17 SystemOutRetryListener 类定义

```
1. public class SystemOutRetryListener implements RetryListener {
2.     public<T>boolean open(RetryContext context,RetryCallback<T> callback) {
3.         System.out.println("SystemOutRetryListener.open()");
4.         return true;
5.     }
6.
7.     public <T> void close(RetryContext context, RetryCallback<T> callback,
8.         Throwable throwable) {
9.         System.out.println("SystemOutRetryListener.close()");
10.    }
11.
12.    public <T> void onError(RetryContext context, RetryCallback<T> callback,
13.        Throwable throwable) {
14.        System.out.println("SystemOutRetryListener.onError()");
15.    }
16. }
```

本拦截器的实现仅打印了当前执行操作的日志，没有具体的业务信息，开发人员可以根据需要完成自己的业务逻辑功能。

10.2.4 重试模板

Spring Batch 框架为面向批的操作提供了自动重试的能力，如果作业步的实现是自定义的 Tasklet，Spring Batch 框架提供了一组方便易用的重试模板 RetryTemplate，使用重试模板可以方便地完成重试功能。目前框架中面向 Chunk 的重试功能同样也是使用 RetryTemplate 来完成的。

通过重试模板可以完成无状态的重试、有状态的重试操作。RetryTemplt 类关系图参见图 10-2。

RetryOperations 接口定义了重试操作的基本方法，重试模板实现该接口；RetryTemplt 提供标准的重试操作；RetryCallback 接口定义了具体的需要重试的逻辑，当具体的重试逻辑发生错误时候，会导致该回调实现的操作按照给定的重试策略进行重试；RetryPolicy 接口定义重试策略，可以使用简单的重试策略或者超时策略；BackOffPolicy 接口定义了补偿策略，每次重试发生的时候可以都会调用该业务补偿；RecoveryCallback 接口定义有状态的业务补偿策略，在所有的重试完成之后会调用该接口完成业务恢复功能；RetryState 接口表示重试状态，用来完成有状态的重试。

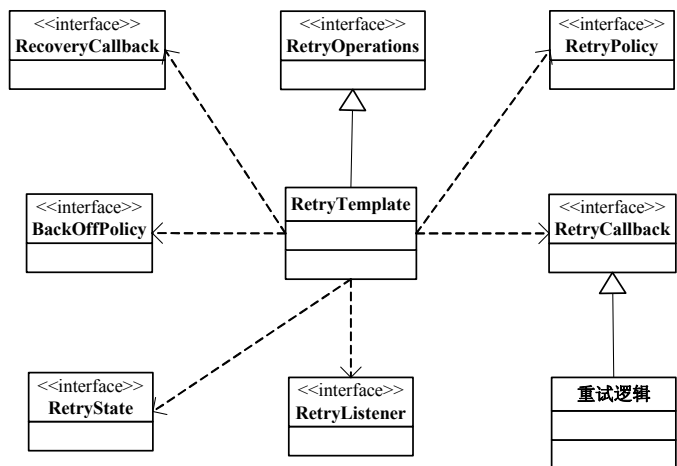


图 10-2 RetryTemplt 类关系图

关键接口、类说明参见表 10-10。

表 10-10 关键接口、类说明

关 键 类	说 明
RetryOperations	重试操作的接口类，定义了如何调用重试的操作，包括无状态的重试、有状态的重试操作； org.springframework.retry.RetryOperations
RetryTemplate	重试模板类，实现 RetryOperations 接口并组装其他接口完成重试功能； org.springframework.retry.support.RetryTemplate
RetryCallback	重试回调接口，当发生重试的时候会多次调用该回调操作，用户可以实现该接口，完成需要重试的业务逻辑； org.springframework.retry.RetryCallback<T>
RetryPolicy	重试策略，可以使用 Spring Batch 框架提供的简单次数重试策略、超时重试策略、或者自定义的重试策略； org.springframework.retry.RetryPolicy
BackOffPolicy	业务补偿操作，每次重试都会触发该接口的 backOff 操作； org.springframework.retry.backoff.BackOffPolicy
RetryListener	重试拦截器，重试发生期间会触发该拦截器的执行，可以定义多个拦截器； org.springframework.retry.RetryListener
RecoveryCallback	重试执行完毕后，会触发恢复回调操作，通常用在有状态的重试中； org.springframework.retry.RecoveryCallback<T>
RetryState	重试状态，用在有状态重试中，可以根据提供的 key 获取重试上下文； org.springframework.retry.RetryState

关键接口说明

接口 `RetryOperations` 定义参见代码清单 10-18。

代码清单 10-18 `RetryOperations` 接口定义

```
1. public interface RetryOperations {
2.     <T> T execute(RetryCallback<T> retryCallback) throws Exception;
3.     <T> T execute(RetryCallback<T> retryCallback,
4.         RecoveryCallback<T> recoveryCallback) throws Exception;
5.     <T> T execute(RetryCallback<T> retryCallback,
6.         RetryState retryState) throws Exception, ExhaustedRetry
7.         Exception;
8.     <T> T execute(RetryCallback<T> retryCallback,
9.         RecoveryCallback<T> recoveryCallback, RetryState retryState)
10.        throws Exception;
11. }
```

`RetryOperations` 接口定义了重试的基本操作，前两个操作是无状态的重试；后两个操作表示有状态的重试。

接口 `RetryCallback` 定义参见代码清单 10-19。

代码清单 10-19 `RetryCallback` 接口定义

```
1. public interface RetryCallback<T> {
2.     T doWithRetry(RetryContext context) throws Exception;
3. }
```

其中，2 行：重试发生的时候会多次调用 `doWithRetry` 操作，可以实现该接口在方法 `doWithRetry` 中实现需要重试的业务逻辑。

接口 `BackOffPolicy` 定义参见代码清单 10-20。

代码清单 10-20 `BackOffPolicy` 接口定义

```
1. public interface BackOffPolicy {
2.     BackOffContext start(RetryContext context);
3.     void backOff(BackOffContext backOffContext) throws BackOffInterrupted
4.         Exception;
5. }
```

接口 `BackOffPolicy` 定义业务补偿操作，`start` 操作在重试过程中仅执行一次，`backOff` 操作在每次重试发生后都会触发该补偿操作。

接口 `RecoveryCallback` 定义参见代码清单 10-21。

代码清单 10-21 `RecoveryCallback` 接口定义

```
1. public interface RecoveryCallback<T> {
2.     T recover(RetryContext ctextext) throws Exception;
3. }
```

`recover` 操作在整个重试操作完成后会被触发。

接口 `RetryState` 定义参见代码清单 10-22。

代码清单 10-22 `RetryState` 接口定义

```
1. public interface RetryState {
2.     Object getKey();
3.     boolean isForceRefresh();
4.     boolean rollbackFor(Throwable exception);
5. }
```

`RetryState` 提供了有状态重试的能力，`getKey` 操作提供了重试逻辑的唯一标识，使用该 `key` 从 `RetryContext` 缓存中获取 `RetryContext`；`isForceRefresh` 操作定义是否每次重新刷新，如果是重新刷新则下次获取到的是一个新的 `RetryContext`；`rollbackFor` 定义哪些类型的异常需要进行回滚操作，如果发生回滚则不会进行重试操作。

具有重试功能的 Tasklet

接下来我们使用重试模板提供的 API 来实现自定义的 `Tasklet`，该 `Tasklet` 具有重试的功能。我们实现信用卡账单 `CreditBillTasklet`，内部使用使用重试模板，使得 `CreditBillTasklet` 具有错误重试的功能。

`CreditBillTasklet` 的实现类参见代码清单 10-23。

完整代码参见：`com.juxtapose.example.ch10.retry.template.CreditBillTasklet`。

代码清单 10-23 `CreditBillTasklet` 类实现

```
1. public class CreditBillTasklet implements Tasklet {
2.     public RepeatStatus execute(StepContribution contribution,
3.         ChunkContext chunkContext) throws Exception {
4.         RetryCallback<String> retryCallback = new DefaultRetryCallback();
5.         RetryListener[] listeners = new RetryListener[]{new CountRetry
6.             Listener()};
7.         SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
8.         retryPolicy.setMaxAttempts(3);
9.         RetryTemplate template = new RetryTemplate();
10.        template.setRetryPolicy(retryPolicy);
11.        template.setListeners(listeners);
12.        template.execute(retryCallback);
13.        return RepeatStatus.FINISHED;
14.    }
```

其中，4 行：定义重试的逻辑操作，`DefaultRetryCallback` 定义了具体的业务实现，当错误发生的时候，`DefaultRetryCallback` 会被执行重试操作。

5 行：定义重试的拦截器，使用 `template.setListeners(listeners)` 注入到重试模板中。

6~7 行：定义重试策略，最大重试次数设置为 3 次。

11 行：使用重试模板执行重试回调操作。

DefaultRetryCallback 实现接口 RetryCallback，实现类中定义了需要重试的逻辑，doWithRetry 中的逻辑发生错误时候，根据重试模板定义的重试策略会重复执行 doWithRetry 操作，直到不再满足重试策略。

DefaultRetryCallback 的实现参见代码清单 10-24。

完整代码参见：`com.juxtapose.example.ch10.retry.template.DefaultRetryCallback`。

代码清单 10-24 DefaultRetryCallback 类定义

```
1. public class DefaultRetryCallback implements RetryCallback<String> {
2.     .....
3.
4.     public String doWithRetry(RetryContext context) throws Exception {
5.         Integer count = (Integer)context.getAttribute("count");
6.         if(count == null){
7.             count = new Integer(0);
8.         }
9.         count++;
10.        context.setAttribute("count", count);
11.        Thread.sleep(sleepTime);
12.        throw new RuntimeException("Mock make exception on business logic.");
13.    }
14. }
```

其中，4~13 行：执行实际的业务逻辑操作，最后模拟每次调用都发生 RuntimeException 异常；注意本处使用了重试上下文 RetryContext，在有状态的重试操作中，可以使用重试上下文在多次重试期间进行数据的共享。

接下来在 Job 文件中配置 CreditBillTasklet，具体参见代码清单 10-25。

完整配置文件参见：`/ch10/job/job-step-retry-tasklet.xml`。

代码清单 10-25 配置 CreditBillTasklet

```
1. <job id="retryTaskletJob">
2.     <step id="retryTaskletStep">
3.         <tasklet ref="retryTasklet"></tasklet>
4.     </step>
5. </job>
6.
7. <bean:bean id="retryTasklet"
8.     class="com.juxtapose.example.ch10.retry.template.
9.         CreditBillTasklet" />
```

2 行：声明自定义的 Tasklet，本身具有重试错误的能力，最大重试次数为 3 次。

补偿策略 BackOffPolicy

重试模板提供了补偿策略，当执行重试的时候，可以为每次重试执行补偿动作，框架提

供了接口 `BackOffPolicy` 来实现该功能。

我们使用如下的示例来验证补偿操作。在重试发生的时候使用计数器来验证功能，`CountRetryListener` 在发生重试异常的时候计数器加 1（参见代码清单 10-26），在补偿策略实现 `DefaultBackoffPolicy` 中计数器减 1（参见代码清单 10-27）。

`CountRetryListener` 拦截器实现参见代码清单 10-26。

完整代码参见：`com.juxtapose.example.ch10.retry.template.CountRetryListener`。

代码清单 10-26 `CountRetryListener` 类定义

```
1. public class CountRetryListener implements RetryListener {
2.
3.     public<T>boolean open(RetryContext context,RetryCallback<T> callback) {
4.         return true;
5.     }
6.
7.     public <T> void close(RetryContext context, RetryCallback<T> callback,
8.         Throwable throwable) {
9.
10.    public <T> void onError(RetryContext context, RetryCallback<T> callback,
11.        Throwable throwable) {
12.        CountHelper.increment();
13.        System.out.println("CountRetryListener.onError().");
14.    }
15. }
```

其中，12 行：拦截器中每次发生重试的时候，计数器加 1；

`DefaultBackoffPolicy` 的实现参见代码清单 10-27。

完整代码参见：`com.juxtapose.example.ch10.retry.template.DefaultBackoffPolicy`。

代码清单 10-27 `DefaultBackoffPolicy` 类定义

```
1. public class DefaultBackoffPolicy implements BackOffPolicy {
2.     public BackOffContext start(RetryContext context) {
3.         BackOffContextImpl backOffContext = new BackOffContextImpl
4.             (context);
5.         return backOffContext;
6.     }
7.
8.     public void backOff(BackOffContext backOffContext)
9.         throws BackOffInterruptedException {
10.        Assert.assertNotNull(((BackOffContextImpl)backOffContext).
11.            getRetryContext().getAttribute("count"));
12.        CountHelper.decrement();
13.    }
```

其中，2~5 行：实现 start 操作，这里根据重试上下文 RetryContext 生成补偿上下文，并持有 RetryContext 的句柄。

7~12 行：每次执行业务的补偿后，计数器减 1。

补偿上下文 BackOffContextImpl 实现接口 BackOffContext，具体实现参见代码清单 10-28。

完整代码参见：`com.juxtapose.example.ch10.retry.template.BackOffContextImpl`。

代码清单 10-28 BackOffContextImpl 类定义

```
1. public class BackOffContextImpl implements BackOffContext {
2.     private RetryContext retryContext;
3.
4.     public BackOffContextImpl() {}
5.     public BackOffContextImpl(RetryContext retryContext) {
6.         this.retryContext = retryContext;
7.     }
8.
9.     public RetryContext getRetryContext() {
10.         return retryContext;
11.     }
12.     public void setRetryContext(RetryContext retryContext) {
13.         this.retryContext = retryContext;
14.     }
15. }
```

补偿上下文持有重试上下文对象，在有状态的重试中可以通过这种方式在每次补偿动作发生的时候获取重试上下文 RetryContext。

使用 JUnit 单元测试来验证补偿策略，参见代码清单 10-29。

完整代码参见：`test.com.juxtapose.example.ch10.RetryTemplateTestCase`。

代码清单 10-29 JUnit 单元测试来验证补偿策略

```
1. @Test
2. public void testBackoffPolicy() {
3.     RetryCallback<String> retryCallback = new DefaultRetryCallback();
4.     SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
5.     retryPolicy.setMaxAttempts(3);
6.     RetryListener[] listeners = new RetryListener[] { new CountRetryListener() };
7.     BackOffPolicy backOffPolicy = new DefaultBackoffPolicy();
8.     RetryTemplate template = new RetryTemplate();
9.     template.setRetryPolicy(retryPolicy);
10.    template.setListeners(listeners);
11.    template.setBackOffPolicy(backOffPolicy);
12.    try {
13.        template.execute(retryCallback);
14.        Assert.assertFalse(true);
15.    }
```



```

15.     } catch (Exception e) {
16.         Assert.assertTrue(true);
17.         Assert.assertEquals(1, CountHelper.getCount());
18.     }
19. }

```

其中，7 行：定义补偿策略。

11 行：将补偿策略设置到重试模板中，在执行重试操作时候会触发补偿策略。

17 行：断言执行补偿策略后，计数器的值保持为 1；这里为什么不是 0，这是因为补偿策略只有在异常被抛出之前执行，最后一次的重试导致了失败，不会触发补偿策略的执行。

有状态重试

重试模板提供了有状态重试的功能，需要使用 `RetryState` 作为参数传入到重试模板。

`RetryState` 的关键作用是提供缓存 `RetryContext` 的 key，定义那些异常不需要重试而是执行回滚操作。通过筛选器 `Classifier` 根据异常类型返回 true 或者 false 来实现后者的功能。

代码清单 10-30 展示了如何使用有状态的重试操作。

完整代码参见：`test.com.juxtapose.example.ch10.RetryTemplateTestCase`。

代码清单 10-30 `RetryTemplateTestCase` 类定义

```

1.  @Test
2.  public void testSimpleRetryPolicyStatefull(){
3.      RetryCallback<String> retryCallback = new DefaultRetryCallback();
4.      SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
5.      retryPolicy.setMaxAttempts(3);
6.      RetryListener[] listeners=new RetryListener[]{new CountRetryListener()};
7.      BackOffPolicy backOffPolicy = new DefaultBackoffPolicy();
8.      RetryTemplate template = new RetryTemplate();
9.      template.setRetryPolicy(retryPolicy);
10.     template.setListeners(listeners);
11.     template.setBackOffPolicy(backOffPolicy);
12.     @SuppressWarnings({ "unchecked", "rawtypes" })
13.     Classifier<? super Throwable, Boolean> classifier =
14.         new ClassifierSupport(Boolean.FALSE);
15.     RetryState retryState = new DefaultRetryState("key", false,
16.         classifier);
17.     try {
18.         template.execute(retryCallback, retryState);
19.         Assert.assertFalse(true);
20.     } catch (Exception e) {
21.         Assert.assertTrue(true);
22.         Assert.assertEquals(1, CountHelper.getCount());
23.     }
24. }

```

其中，13~14 行：定义异常筛选类，本处使用框架提供的 `Support` 类，对所有类型的异常都会返回 `false`，`false` 表示重试逻辑发生任何异常的时候都不会导致 `rollback` 操作，而是执行重试操作。

15 行：定义 `RetryState`，本节使用“key”来作为唯一的關鍵字，将 `RetryContext` 通过关键字“key”缓存到 `Map` 中，后续每次重试的时候都能通过该关键字获取 `RetryContext`。

10.3 重启 Restart

即便再健壮的 `Job`，解决了 `Skip`、`Retry` 的问题，也有可能最终执行 `Job` 失败。在 `Job` 失败的场景下是让用户重头再次执行 `Job` 还是能够从上次 `Job` 失败的地点重新执行 `Job`？`Spring Batch` 框架提供了重启 `Job` 的功能，包括重启 `Job`、`Step` 支持重启、重启已经完整的 `Step`、禁止 `Step` 重启、限制重启次数等功能。

`Job` 的重启是通过多个 `Job Execution` 来完成的，每个 `Job Instance` 可以有多个 `Job Execution`，`Job` 执行器负责完成 `Job` 实例。

一个典型的任务重启的流程参见图 10-3。

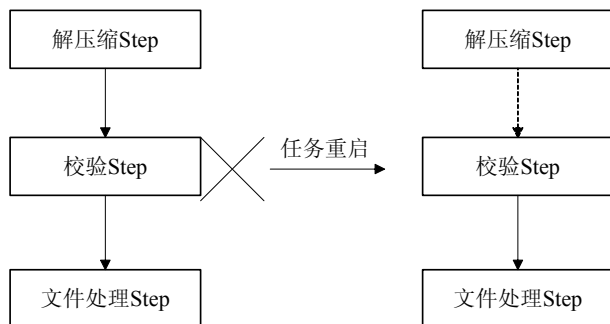


图 10-3 典型的任务重启的流程

第一次 `Job` 执行过程中校验 `Step` 出错，接下来重启 `Job` 会重上次失败的点进行重新启动 `Job`。`Job` 的两次执行过程中，对应同一个 `Job` 的实例，但是执行器是不同的两个执行器。

`Job` 定义、`Job` 实例、`Job` 执行器三者的关系参见图 10-4。

10.3.1 重启 Job

`Spring Batch` 框架对重启 `Job` 有如下的规则可以遵守。

- (1) 只能重启状态为失败的 `Job` 实例。
- (2) 任何 `Job` 失败的实例都可以被重新执行。
- (3) 重新执行 `Job` 的时候，会从上上次执行失败的点重新开始执行，而不是从头开始执行。
- (4) 已经完成的 `Step`，通过特殊的标识也可以被重新执行。

(5) 一个失败的 Job 可以被不断的执行，取费有重启次数的限制。

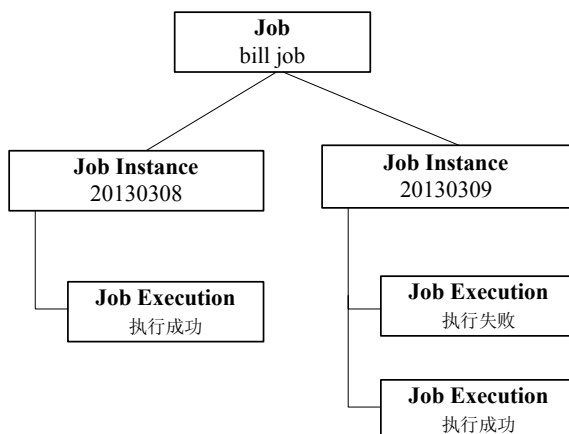


图 10-4 Job 定义、Job 实例、Job 执行器三者的关系

重启 Job 的几个关键属性包括 restartable、allow-start-ifcomplete、start-limit，具体描述参见表 10-11。

表 10-11 重启属性说明

属 性	所属节点	说 明	默认值
restartable	Job	定义当前作业是否支持重启，默认值是 true，表示支持重启，如果不需要重启，需要显示设置为 false	True
allow-start-ifcomplete	tasklet	是否允许完成(状态为"COMPLETED")的 Step 重新启动	false
start-limit	tasklet	Step 能够启动的最大次数 超过最大次数后会抛出异常	Integer.MAX_VALUE

代码清单 10-31 展示了如何设置 Job 是可以重新启动的。

代码清单 10-31 设置 Job 可以重新启动

```

1.     <job id="billJob" restartable="true">
2.         <step id="billStep">
3.             <tasklet transaction-manager="transactionManager">
4.                 <chunk reader="csvItemReader" writer="csvItemWriter"
5.                     processor="creditBillProcessor" commit-interval="2">
6.                     </chunk>
7.                 </tasklet>
8.             </step>
9.         </job>
  
```

其中，1 行：设置当前的 Job 可以重新启动，如果设置属性 `restartable` 的值为 `false`，则该 Job 不支持重新启动。

10.3.2 启动次数限制

默认情况下，作业实例可以无限次地重复启动。在有些场景下需要限制作业实例的启动次数，例如在执行任务分区（`particular`）的 Step 中，需要对分区的 Step 限制启动次数为 1 次，因为在数据错误的情况下，多次重启 Step 没有任何意义。当然，读者可以找到更多的场景来使用限制任务重启的次数。限制启动次数示例参见代码清单 10-32。

代码清单 10-32 配置启动次数限制

```
1. <step id="startLimitStep">
2.   <tasklet start-limit="2">
3.     <chunk reader="reader" processor="processor" writer="writer"/>
4.   </tasklet>
5. </step>
```

定义 `startLimitStep` 仅能启动二次，第三次启动的时候会抛出异常。`start-limit` 默认值是 `Integer.MAX_VALUE`，表示任务可以无限次地启动。

10.3.3 重启已完成的任务

通常情况下已经完成的 Step 默认是不会自动重新启动的，可以通过属性 `allow-start-if-complete` 来设置已经完成的 Step 在重启的时候可以再次被执行，具体的效果图参见图 10-5。

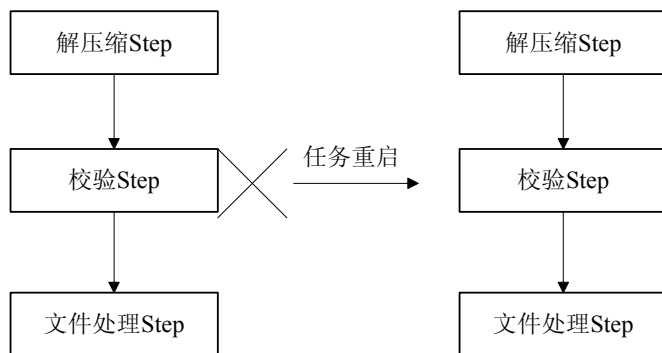


图 10-5 重启已经完成的 Step

默认情况下，Job Instance 重新启动的时候，已经完成的任务不会再次被执行。单在某些特殊场景下，已经完成的 Step 在任务重启的时候需要再次执行，可以通过属性 `allow-start-if-complete` 来设置。示例配置文件参见代码清单 10-33。

代码清单 10-33 配置可重启的完成任务

```
1.     <job id="conditionalJob" >
2.         <step id="decompressStep" parent="abstractDecompressStep"
3.             next="verifyStep" >
4.                 <tasklet ref="decompressTasklet" allow-start-if-complete="true"/>
5.             </step>
6.             .....
7.         <step id="cleanStep">
8.             <tasklet ref="cleanTasklet" />
9.         </step>
10.    </job>
```

其中，4 行：定义解压缩 Step 即时在状态为完成的情况下，仍然可以再次被执行。

扩展 Job、并行处理

前面我们基本掌握了如何使用 Spring Batch 开发批处理任务，面向批处理的作业通常涉及大数据的处理、高的资源消耗。如何实现性能可靠的、可扩展的作业是一个比较高的挑战工作。在介绍如何开发高扩展性的 Job 之前，我们首先了解下软件系统的可扩展性。

11.1 可扩展性

可扩展性（可伸缩性）是一种对软件系统计算处理能力的设计指标，高可扩展性代表一种弹性，在系统扩展成长过程中，软件能够保证旺盛的生命力，通过很少的改动甚至只是硬件设备的添置，就能实现整个系统处理能力的线性增长，实现高吞吐量和低延迟高性能。

软件系统的扩展通常可以通过如下的两种方式实现，垂直扩展（参见图 11-1）、水平扩展（参见图 11-2）。

垂直扩展是通过升级原有的服务器或者为当前的应用更换更强大的硬件来实现系统处理能力的增强。比如为当前的服务器增加更大的内存、存储、处理器资源等。垂直扩展通常要求提供更强大的服务器资源来达到增加软件处理的能力；通过更换更强的服务器可以方便地实现软件系统的可扩展性，但是这种便利性同时也有较大的局限性，毕竟这种处理能力的增加是有限的，因为单个服务器的处理能力毕竟最终是有限的。

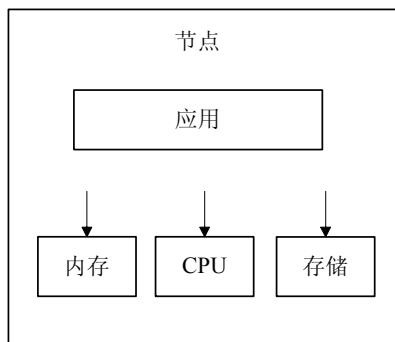


图 11-1 垂直扩展

水平扩展指的是通过增加更多的服务器来分散负载，可以将多个服务器从逻辑上看成一个实体，从而实现存储能力和计算能力的扩展。比如，可以简单地通过聚类或负载均衡策略，通过增加多个服务器来加快整个逻辑实体的运行速度及性能。

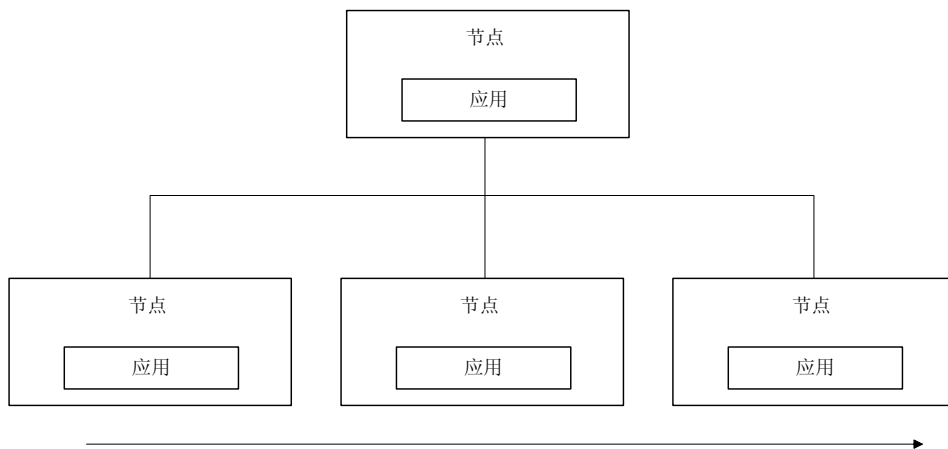


图 11-2 水平扩展

Spring Batch 框架提供了多种提高 Job 并行处理、扩展性的方式。通常情况下只需要调整 Job 的配置就可以达到扩展处理 Job 的目的。需要注意的是框架提供了在 Step 级别进行任务的扩展能力。

框架提供的扩展能力包括如下的四种模式，具体参见表 11-1。

表 11-1 Spring Batch 框架提供扩展能力的四种模式

扩展模式	Local/Remote	说 明
Multithreaded step 多线程作业步	Local	Step 可以使用多线程执行（通常一个 Step 是由一个线程执行的）
Parallel step 并行作业步	Local	Job 执行期间，不同的 Step 并行处理，由不同的线程执行（通常 Job 的 Step 都是顺序执行，且由同一个线程执行的）
Partitioning step 分区作业步	Local/Remote	通过将任务进行分区，不同的 Step 处理不同的任务数据达到提高 Job 效率的功能
Remote chunking 远程任务	Remote	将任务分发到远程不同的节点进行并行处理，提高 Job 的处理速度和效率

接下来我们将学习上面的四种扩展模式。

11.2 多线程 Step

批处理框架在 Job 执行时默认使用单个线程完成任务的执行，同时框架提供了线程池的支持，可以在 Step 执行时候进行并行处理，这里的并行是指同一个 Step 使用线程池进行执行，同一个 Step 被并行地执行。使用 tasklet 的属性 task-executor 可以非常容易地将普通的 Step 变成多线程 Step。配置多线程 Step 属性具体说明参见表 11-2。

表 11-2 配置多线程 Step 属性说明

属 性	说 明	默 认 值
task-executor	任务执行处理器，定义后表示采用多线程执行任务，需要考虑多线程执行任务时候的安全性	
throttle-limit	最大使用线程池的数目	6

11.2.1 配置多线程 Step

代码清单 11-1 给出了配置多线程 Step 的示例。

完整配置参见文件：/ch11/job/job-multithread.xml。

代码清单 11-1 配置多线程 Step 示例

```

1. <job id="multiThreadJob">
2.     <step id="multiThreadStep">
3.         <tasklet task-executor="taskExecutor" throttle-limit="6">
4.             <chunk reader="reader" processor="processor" writer="writer"
               commit-interval="2"/>
5.         </tasklet>
6.     </step>
7. </job>

```

其中，3 行：属性 task-executor 定义执行 Step 需要的线程池，属性 throttle-limit 用于限制能使用的最大的线程数目；线程池的具体配置参见代码清单 11-2。

线程池的配置

代码清单 11-2 线程池的配置

```

1. <bean:bean id="taskExecutor"
2.     class="org.springframework.scheduling.concurrent.
       ThreadPoolTaskExecutor">
3.     <bean:property name="corePoolSize" value="5"/>
4.     <bean:property name="maxPoolSize" value="15"/>
5. </bean:bean>

```

为了便于展示不同的线程执行上面的 Step，我们在 Read、Write 的实现中增加了打印当前线程的代码，执行上面的 Job，在控制台能够打印出代码清单 11-3 的片段。

代码清单 11-3 多线程 Step 执行控制台输出结果

```

1. Read:0;Job Read Thread name: taskExecutor-1
2. Read:1;Job Read Thread name: taskExecutor-1
3. Write begin:0,1,Write end!!Job Write Thread name: taskExecutor-1
4. Read:2;Job Read Thread name: taskExecutor-3
5. Read:3;Job Read Thread name: taskExecutor-3
6. Write begin:2,3,Write end!!Job Write Thread name: taskExecutor-3

```



```
7. Read:4;Job Read Thread name: taskExecutor-2
8. Read:5;Job Read Thread name: taskExecutor-2
9. Write begin:4,5,Write end!!Job Write Thread name: taskExecutor-2
10. Read:6;Job Read Thread name: taskExecutor-4
11. Read:7;Job Read Thread name: taskExecutor-4
12. Write begin:6,7,Write end!!Job Write Thread name: taskExecutor-4
13. Read:8;Job Read Thread name: taskExecutor-5
14. Read:9;Job Read Thread name: taskExecutor-5
15. Write begin:8,9,Write end!!Job Write Thread name: taskExecutor-5
16. Read:10;Job Read Thread name: taskExecutor-1
17. Read:11;Job Read Thread name: taskExecutor-1
18. Write begin:10,11,Write end!!Job Write Thread name: taskExecutor-1
19. Read:12;Job Read Thread name: taskExecutor-3
20. Read:13;Job Read Thread name: taskExecutor-3
21. Write begin:12,13,Write end!!Job Write Thread name: taskExecutor-3
22. ....
```

可以看出在 Step 执行期间共有 5 个不同的线程进行了作业 Step 处理;另外大家需要注意在 Step 的读、处理、写执行时,是在一个完整的线程中进行处理的;另外一个需要注意的是同一个线程在进行任务处理的时候,其处理的数据是不连续的,例如线程 taskExecutor-1 在处理完数据 0、1 之后,接下来处理的数据为 10,11。

说明:

在多线程 Step 中为了保证代码处理的正确性,要求所有在多线程 Step 中处理的对象和操作必须是线程安全的;简单期间任何无状态的处理操作都是线程安全的,对于有状态的操作可以通过特殊的处理变成线程安全的操作。但是 Spring Batch 框架提供的大部分的 ItemReader、ItemWriter 等操作都是线程不安全的,最主要的原因在于 ItemReader、ItemWriter 提供了可重启特性的支持,在运行期间保存了大量的运行期状态导致了 ItemReader、ItemWriter 操作均是有状态的,不能直接运用在多线程步中。

11.2.2 线程安全性

我们首先学习一下什么是线程安全的,如果代码所在的进程中有多个线程在同时运行,而这些线程可能会同时运行这段代码;如果每次运行结果和单线程运行的结果是一样的,而且其他变量的值也和预期的是一样的,我们称之为代码是线程安全的。

在 Java 领域线程安全问题通常是全局变量或者静态变量引起的,若每个线程中对全局变量、静态变量只有读操作,而无写操作,一般来说,这个全局变量是线程安全的;若有多个线程同时执行写操作,则需要考虑线程同步,否则的话就可能影响线程安全。

我们以 Java 中的 ArrayList 举例,向 List 中增加元素(假设目前 List 中有 0 个元素),来描述线程的安全性。

单线程:在单线程运行的情况下,向 List 中增加一个元素,当前元素的位置为 0,共有 1

个元素；继续增加一个元素，因为是串行执行，结果是当前元素的位置为 1，共有 2 个元素。

多线程：假设有 2 个线程（分别是线程 1、线程 2）执行，线程 1 先将元素存放在位置 0，此时 CPU 调度线程 1 暂停，线程 2 得到运行的机会，线程 2 也向此 ArrayList 添加元素，因为此时 Size 仍然等于 0，所以线程 2 也将元素存放在位置 0；然后线程 1 和线程 2 都继续运行，都增加 Size 的值；最终的结果是 List 的 Size 为 2，但实际上只有 1 个元素，这就导致了 List 是线程不安全的。

Spring Batch 框架中提供的大量 ItemReader、ItemWriter 通常是线程不安全的，因为大多数的 ItemReader、ItemWriter 是有状态的，导致了无法直接在多线程 Step 中直接使用这些基础设施。通常可以通过 synchronized 来实现并发控制，或者通过业务规则来实现线程安全的 ItemReader、ItemWriter 等组件。

接下来我们将学习如何实现线程安全的 ItemReader。

11.2.3 线程安全 Step

ItemReader、ItemWriter 在运行过程中保存了部分状态信息，用于支撑 Step 的重启操作。这些状态导致是非线程安全的组件。接下来我们介绍如何实现线程安全的 ItemReader。本节我们以 JdbcCursorItemReader 为例来实现线程安全的组件。

JdbcCursorItemReader 不是线程安全的，不能直接用于多线程 Step 中。一个简单的方法是对 read 操作使用关键字 synchronized 进行同步执行，这样可以保证在执行过程中保证线程安全性。使用 synchronized 会造成多线程读取时候的阻塞，但我们考虑下载批处理的作业中读操作通常是非常轻量级的，更多的处理器资源都消耗在处理操作、写操作上。因此对读操作的同步等待通常不会造成性能瓶颈。

如果在实际的业务开发中同步读导致性能问题，我们可以采取数据分区的功能来解决。具体分区 Step 的实现请参见 11.4 节分区 Step。

为了使 JdbcCursorItemReader 保证线程安全的，我们实现一个同步的 SynchronizedItemReader，参见代码清单 11-4，对 read 操作使用 synchronized 关键字。

完整代码参见：com.juxtapose.example.ch11.multithread.SynchronizedItemReader。

代码清单 11-4 SynchronizedItemReader 类定义

```
1. public class SynchronizedItemReader implements ItemReader<CreditBill>,
   ItemStream{
2.
3.     private ItemReader<CreditBill> delegate;
4.
5.     public synchronized CreditBill read() throws Exception {
6.         CreditBill creditBill = delegate.read();
7.         return creditBill;
8.     }
9. }
```

```

10.     public void close() throws ItemStreamException {
11.         if (this.delegate instanceof ItemStream) {
12.             ((ItemStream)this.delegate).close();
13.         }
14.     }
15.
16.     public void open(ExecutionContext context) throws ItemStreamException {
17.         if (this.delegate instanceof ItemStream) {
18.             ((ItemStream)this.delegate).open(context);
19.         }
20.     }
21.
22.     public void update(ExecutionContext context) throws
ItemStreamException {
23.         if (this.delegate instanceof ItemStream) {
24.             ((ItemStream)this.delegate).update(context);
25.         }
26.     }
27.     .....
28. }

```

其中，3 行：代理给定的 `ItemReader`，此处我们使用 `JdbcCursorItemReader`。

5 行：给定的 `read` 操作使用 `synchronized`，保证多线程并发时刻按照处理器分配的顺序执行代理类的 `read` 操作，保证读操作的线程安全特性。

10~26 行：实现 `ItemStream` 接口的 `open`、`close`、`update` 操作。

配置线程安全的 `ItemReader` 的 `Job`，配置代码参见代码清单 11-5。

完整配置文件参见：`/ch11/job/job-multithread-db.xml`。

代码清单 11-5 配置线程安全的 `ItemReader` 的 `Job`

```

1.     <job id="dbSynchronizedJob">
2.         <step id="dbSynchronizedStep">
3.             <tasklet task-executor="taskExecutor" throttle-limit="6">
4.                 <chunk reader="creditBillItemReader"
5.                     processor="creditBillProcessor"
6.                         writer="jdbcSetterItemWriter" commit-interval="2">
7.                     </chunk>
8.                 </tasklet>
9.             </step>
10.        </job>
11.
12.        <bean:bean id="creditBillItemReader"
13.            class="com.juxtapose.example.ch11.multithread.
14.                SynchronizedItemReader">
15.            <bean:property name="delegate" ref="jdbcItemReader" />
16.        </bean:bean>

```

```

15.
16.     <bean:bean id="jdbcItemReader" scope="step"
17.         class="org.springframework.batch.item.database.
            JdbcCursorItemReader" >
18.         <bean:property name="dataSource" ref="dataSource"/>
19.         <bean:property name="sql"
20.             value="select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS from
                t_credit"/>
21.         <bean:property name="verifyCursorPosition" value="false"></bean:
            property>
22.         <bean:property name="saveState" value="false"></bean:property>
23.     </bean:bean>

```

其中，1~9 行：定义 Job，读组件使用线程安全的读，使用上面实现的 SynchronizedItem Reader。

11~14 行：定义线程安全的读组件，代理 JDBC 的数据库读组件。

16~23 行：定义 JDBC 提供的标准的数据库读，需要注意两个属性 saveState 和 verifyCursorPosition 都设置为 false。

11.2.4 可重启的线程安全 Step

上面我们实现了线程安全的读组件，但是上面的实现并不支持重启操作，当执行失败的时候因为没有保存当前读取的状态数据导致无法知道哪些数据已经读取成功，哪些是未读取的。接下来我们学习如何实现可重启的线程安全的读组件。

数据库读取组件 JdbcCursorItemReader，我们可以友好地设计数据库表，在读取的表中增加一个字段 Flag，用于标识当前的记录是否已经读取并处理成功，如果处理成功则标识 Flag=true，等下次重新读取的时候，对于已经成功读取且处理成功的记录直接跳过处理。

可重启的线程安全的 Step 处理逻辑图参见图 11-3。

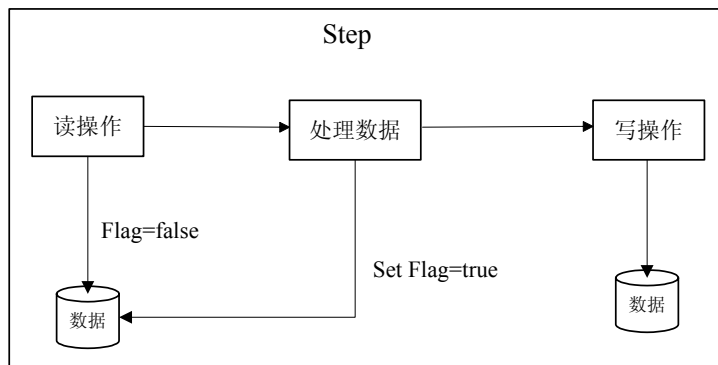


图 11-3 可重启的线程安全的 Step 处理逻辑

需要在数据处理成功后，通过 JDBCTemplate 模板将 Flag 的标识值写入到对应的数据库记录中。

设计新的数据库表 t_credit，增加 Flag 字段用于标识该行记录是否正确的处理过，代码清单 11-6 给出了完整的数据库创建脚本。

完整数据库表参见文件：/ch11/db/create-tables-mysql.sql。

代码清单 11-6 数据库创建脚本

```
1. CREATE TABLE t_credit
2.     (ID VARCHAR(64),
3.       ACCOUNTID VARCHAR(20),
4.       NAME VARCHAR(10),
5.       AMOUNT NUMERIC(10,2),
6.       DATE VARCHAR(20),
7.       ADDRESS VARCHAR(128),
8.       FLAG VARCHAR(10),
9.       primary key (ID)
10.    )
11.    ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

其中，8 行：字段 FLAG 用于标识当前记录是否执行成功。

可重启的线程安全 Step 的 Job 配置参见代码清单 11-7。

完整配置参见文件：/ch11/job/job-multithread-db.xml。

代码清单 11-7 配置可重启的线程安全 Step 的 Job

```
1.     <job id="dbRestartSynchronizedJob">
2.         <step id="dbRestartSynchronizedStep">
3.             <tasklet task-executor="taskExecutor" throttle-limit="6">
4.                 <chunk reader="creditBillRestartItemReader"
5.                     processor="creditBillRestartProcessor"
6.                     writer="jdbcSetterItemWriter" commit-interval="2">
7.                     </chunk>
8.                 </tasklet>
9.             </step>
10.        </job>
11.
12.    <bean:bean id="creditBillRestartItemReader"
13.        class="com.juxtapose.example.ch11.multithread.
14.        SynchronizedItemReader">
15.        <bean:property name="delegate" ref="jdbcRestartItemReader" />
16.    </bean:bean>
17.    <bean:bean id="jdbcRestartItemReader" scope="step"
18.        class="org.springframework.batch.item.database.
19.        JdbcCursorItemReader" >
20.        <bean:property name="dataSource" ref="dataSource"/>
```

```

19.         <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
20.         DATE,ADDRESS from t_credit where flag=false"/>
21.         .....
22.     </bean:bean>

```

其中，1~10 行：定义可重启的线程安全的 Job。

5 行：特殊的数据处理，对应的实现类参见代码清单 11-8。

12~14 行：定义线程安全的 Step；此处使用上面用到的 `synchronized` 的线程安全组件。

16~22 行：定义 JDBC 的读组件，需要注意在读取的 SQL 中需要根据字段 `flag` 进行过滤，已经处理成功的记录不会被再次被处理。

接下来我们学习如何配置处理器，在处理阶段进行数据库记录的修改，需要重新实现处理类 `CreditBillProcessor`，实现代码参见代码清单 11-8。

完成类实现参见文件：`com.juxtapose.example.ch11.multithread.CreditBillProcessor`。

代码清单 11-8 `CreditBillProcessor` 类定义

```

1. public class CreditBillProcessor implements
2.     ItemProcessor<CreditBill, DestinationCreditBill> {
3.     JdbcTemplate jdbcTemplate = null;
4.
5.     public DestinationCreditBill process(CreditBill bill) throws Exception {
6.         .....
7.         if(null != jdbcTemplate){
8.             jdbcTemplate.update("update t_credit set flag=? where id=?",
9.                 "true", bill.getId());
10.        }
11.        return destCreditBill;
12.    }
13. }

```

其中，8 行：使用 `jdbcTemplate` 更新记录的 `Flag` 标识位为 `true`。

用户可以基于上面类似的方案实现自定义的可重启的且线程安全的实现。通常情况下实现上述方案比较复杂，对于大数据处理而言，可以将需要处理的数据进行恰当地分区，交给不同的 Job 或者不同的 Step 进行处理，这些 Job 或者 Step 可以使本地的也可以是远程的；通过恰当的分区可以避免实现复杂的线程安全的读/写组件。

11.3 并行 Step

多线程步提供了多个线程执行一个 Step 的能力，但这种场景在实际的业务中使用的并不是非常多。更多的业务场景是 Job 中不同的 Step 没有明确的先后顺序，可以在执行期间并行地执行。Spring Batch 框架提供了并行 Step 的能力。可以通过 `Split` 元素来定义并行的作业流，

并制定使用的线程池。

并行 Step 的执行关系参见图 11-4 。

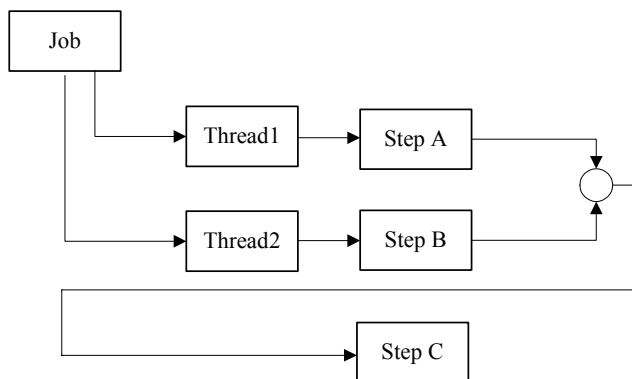


图 11-4 并行 Step 的执行关系

Step A、Step B 两个作业步由不同的线程执行，两者均执行完毕后，Step C 才会被执行。

配置并行执行的 Step 非常的简单，只需要使用 Spring Batch 框架提供的 Split 元素就可以完成，无须额外地编写任何的代码，将复杂度和业务代码的实现进行了有效的隔离，简化了开发难度。具体的如何使用 Split 请参见 9.3 章节的并行 Flow。

并行 Step 提供了在一个节点上横向处理，随着作业处理量的增加，有可能一台节点无法满足 Job 的处理，此时我们可以采用远程 Step 的方式将多个机器节点组合起来完成一个 Job 的处理。我们将在接下来的章节介绍远程 Step。

11.4 远程 Step

远程分块是一个把 step 进行技术分割的工作，不需要对处理数据的结构有明确了解。任何输入源都能够使用单进程读取并在动态分割后作为"块"发送给远程的工作进程。远程进程实现了监听者模式，反馈请求、处理数据，最终将处理结果异步返回。请求和返回之间的传输会被确保在发送者和单个消费者之间。Spring Batch 在 Spring Integration 顶部实现了远程分块的特性。接下来我们将向读者展示如何使用 Spring Integration 的技术实现远程 Step 的功能。

11.4.1 远程 Step 框架

在 Spring Batch 中对远程 Step 没有默认的实现，但是提供了远程 Step 的框架，通过框架可以方便地扩展出远程 Step 的实现。

远程 Step 技术本质上是对 Item 读、写的处理逻辑进行分离；通常情况下读的逻辑放在一个节点进行操作，将写操作分发到另外的节点执行。

远程 Step 的作业情况参见图 11-5。

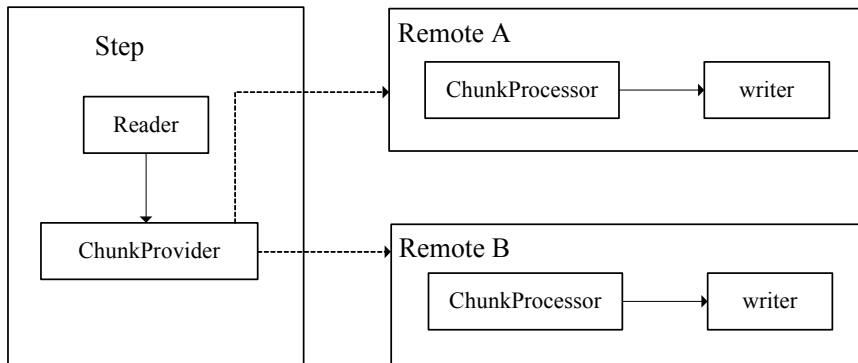


图 11-5 远程 Step 作业逻辑

在 Master 节点，作业步负责读取数据，并将读取的数据通过远程技术发送到指定的远端节点上，进行处理，处理完毕后 Master 负责回收 Remote 端执行的情况。在 Spring Batch 框架中通过两个核心的接口来完成远程 Step 的任务，分别是 ChunkProvider 与 ChunkProcessor。

ChunkProvider: 根据给定的 ItemReader 操作产生批量的 Chunk 操作；接口定义参见代码清单 11-9。

代码清单 11-9 ChunkProvider 接口定义

```
1. public interface ChunkProvider<T> {
2.     Chunk<T> provide(StepContribution contribution) throws Exception;
3.     void postProcess(StepContribution contribution, Chunk<T> chunk);
4. }
```

其中，2 行：操作 provider 产生 Chunk 操作，该 Chunk 操作通过各种远程的技术发送到远端进行执行。

ChunkProcessor: 负责获取 ChunkProvider 产生的 Chunk 操作，执行具体的写逻辑；接口定义参见代码清单 11-10。

代码清单 11-10 ChunkProcessor 接口定义

```
1. public interface ChunkProcessor<I> {
2.     void process(StepContribution contribution, Chunk<I> chunk) throws
        Exception;
3. }
```

其中，2 行：操作 process 负责处理远端获取到的 Chunk。

在 Spring Batch 中对远程 Step 没有默认地实现，但 Spring 中提供了另外一个项目，Spring Batch Integration 项目，将 Spring Batch 框架和 Spring Integration 做了集成，可以通过 Spring Integration 提供的远程能力实现远程 Step。接下来我们将介绍如何通过 Spring Integration (SI) 实现远程 Step。

11.4.2 基于 SI 实现远程 Step

Spring Batch Integration 项目中提供了远程 Step 的能力，关键是利用了 SI 提供的远程队列的能力。

Spring Batch Integration 中实现的远程 Step 的结构图参见图 11-6。

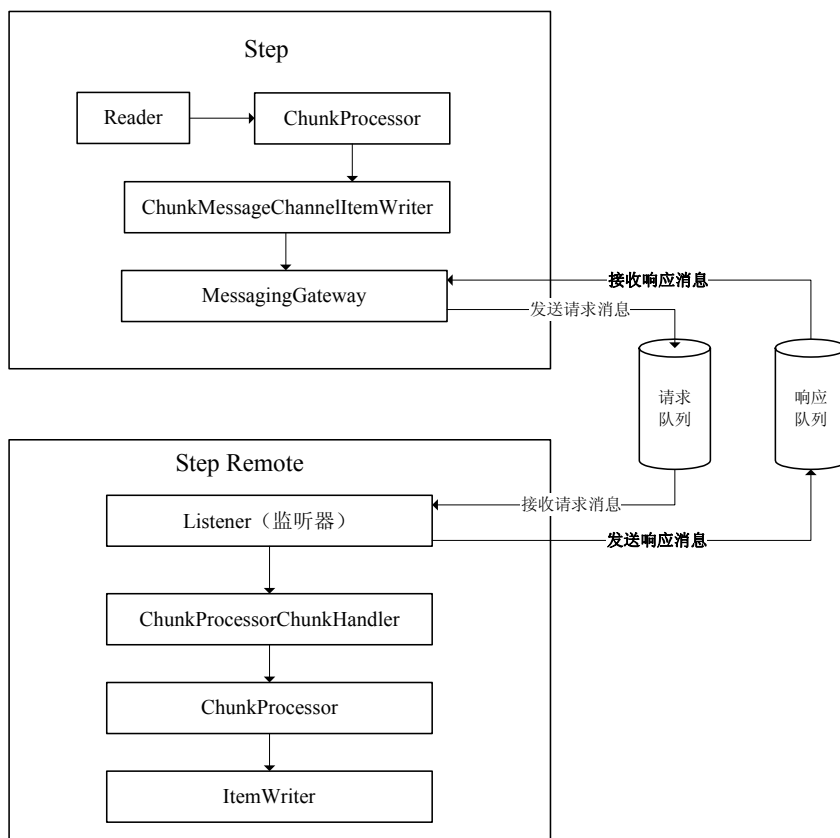


图 11-6 Spring Batch Integration 中实现的远程 Step 的结构图

Step 本地节点负责读取数据，并通过 MessagingGateway 将请求发送到远程 Step 上；远程 Step 提供了队列的监听器，当请求队列中有消息时候获取请求信息并交给 ChunkHandler 负责处理。

接下来我们展示如何配置基于 SI 的远程 Step；如果读者对 SI 不熟悉，可以通过 Spring 官方网站学习该技术，本章节不会具体讲解 SI 的技术。

接下来的章节我们采用的示例是从数据库表 `t_credit` 中读取数据，然后通过远程的方式将 Step 远端执行，将数据写入表 `t_destcredit` 中。远程 Step 数据处理的示意图参见图 11-7。

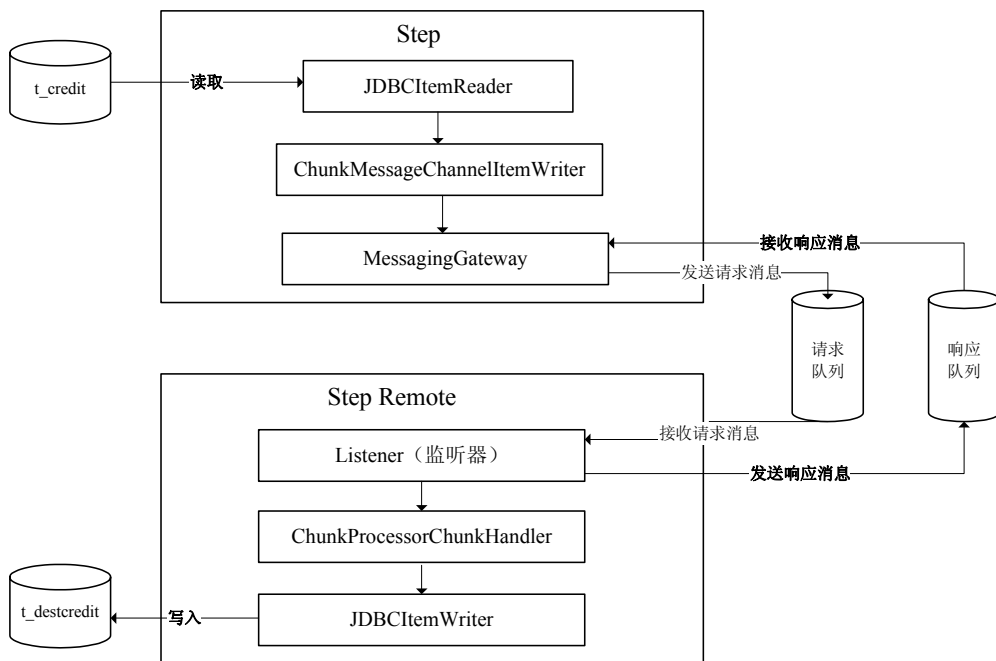


图 11-7 远程 Step 数据处理的示意图

11.4.2.1 配置消息队列

下面定义了需要使用的请求队列、响应队列,以及本地 Step 节点需要的 MessagingGateway,通过 MessagingGateway 可以将请求消息发送到 requests 队列中,并从 replies 队列中获取响应的消息。具体消息队列配置参见代码清单 11-11。

完整配置参见文件: /ch11/job/job-chunk-remote.xml。

代码清单 11-11 配置消息队列

```

1. <bean:bean id="messagingGateway"
    class="org.springframework.integration.core.MessagingTemplate">
2.     <bean:property name="defaultChannel" ref="requests" />
3.     <bean:property name="receiveTimeout" value="1000" />
4. </bean:bean>
5. <int-jms:outbound-channel-adapter connection-factory="connectionFactory"
    channel="requests" destination-name="requests" />
6. <int:channel id="requests" />
7. <int:channel id="incoming" />
8. <int:transformer input-channel="incoming" output-channel="replies"
9.     ref="headerExtractor" method="extract" />
10. <int:channel id="replies" scope="thread">
11.     <int:queue />

```

```

12.     <int:interceptors>
13.         <bean:bean id="pollerInterceptor"
14.             class="org.springframework.batch.integration.
                    chunk.MessageSourcePollerInterceptor">
15.             <bean:property name="messageSource">
16.                 <bean:bean class="org.springframework.integration.
                        jms.JmsDestinationPollingSource">
17.                     <bean:constructor-arg>
18.                         <bean:bean class="org.springframework.jms.
                                core.JmsTemplate">
19.                             <bean:property name="connectionFactory"
                                    ref="connectionFactory" />
20.                             <bean:property name="defaultDestinationName"
                                    value="replies" />
21.                             <bean:property name="receiveTimeout"
                                    value="100" />
22.                         </bean:bean>
23.                     </bean:constructor-arg>
24.                 </bean:bean>
25.             </bean:property>
26.             <bean:property name="channel" ref="incoming"/>
27.         </bean:bean>
28.     </int:interceptors>
29. </int:channel>

```

其中，1~4 行：定义消息网关，负责向 `requests` 队列发送消息，从 `replies` 队列收取消息。

5 行：定义请求队列 `requests` 的 `adapter`。

6~7 行：定义使用的 `channel`。

8~9 行：定义消息转换器。

10~29 行：定义响应 `channel`。

11.4.2.2 配置 AMQ 服务器

接下来的配置定义了 AMQ 的服务器定义，在配置 AMQ 服务器定义时需要在 XML 中指定对应的命名空间 `xmlns:amq="http://activemq.apache.org/schema/core"`。配置 AMQ 服务器参见代码清单 11-12。

完整配置参见文件：`/ch11/job/job-chunk-remote.xml`。

代码清单 11-12 配置 AMQ 服务器

```

1. <bean:bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
2.     <bean:property name="connectionFactory" ref="connectionFactory" />
3.     <bean:property name="receiveTimeout" value="100" />

```

```

4.     <bean:property name="sessionTransacted" value="true" />
5. </bean:bean>
6.
7. <amq:broker useJmx="false" persistent="false" schedulerSupport="false">
8.   <amq:transportConnectors>
9.     <amq:transportConnector uri="tcp://localhost:61616"/>
10.   </amq:transportConnectors>
11. </amq:broker>
12. <amq:connectionFactory id="connectionFactory" brokerURL="tcp://localhost:
    61616"/>

```

其中，1~5 行：定义使用的 JMS 消息模板 `jmsTemplate`。

7~12 行：定义了使用的 AMQ 服务器，默认使用的端口号为 61616。

11.4.2.3 配置本地 Step

本地 Step 负责读取消息，并将对应的 Chunk 通过 JMS 队列的方式发送到远端 Step 执行。本地 Step 配置参见代码清单 11-13。

完整配置参见文件：`/ch11/job/job-chunk-remote.xml`。

代码清单 11-13 配置本地 Step

```

1. <bean:bean id="chunkWriter" scope="step"
2.   class="org.springframework.batch.integration.
        chunk.ChunkMessageChannelItemWriter" >
3.   <bean:property name="messagingOperations" ref="messagingGateway" />
4.   <bean:property name="replyChannel" ref="replies" />
5.   <bean:property name="maxWaitTimeouts" value="10" />
6. </bean:bean>
7.
8. <bean:bean id="chunkHandler"
9.   class="org.springframework.batch.integration.
        chunk.RemoteChunkHandlerFactoryBean">
10.   <bean:property name="chunkWriter" ref="chunkWriter" />
11.   <bean:property name="step" ref="stepRemoteChunk" />
12. </bean:bean>

```

其中，1~6 行：定义 `chunkWriter`，`chunkWriter` 负责将本地 Step 中的读取数据通过 `messagingGateway` 发送端远程 Step 中；`ChunkMessageChannelItemWriter` 是一个 `StepExecutionListener` 类型的拦截器。

8~12 行：`RemoteChunkHandlerFactoryBean` 在创建 `chunkHandler` 过程中，默认注册了 `ChunkProcessor`，与 `chunkWriter` 完成了工作的传递。

11 行：定义远程拦截器调用使用的作业步为 `stepRemoteChunk`。

除了使用 `RemoteChunkHandlerFactoryBean` 创建 `chunkHandler` 外，还可以直接通过

org.springframework.batch.integration.chunk.ChunkProcessorChunkHandler 声明。下面给出直接使用 ChunkProcessorChunkHandler 而不是 RemoteChunkHandlerFactoryBean 的方式创建 chunkHandler，具体参见代码清单 11-14。

完整的配置文件参见：ch11/job/job-chunk-remote-other.xml。

代码清单 11-14 配置本地 Step（使用 ChunkProcessorChunkHandler）

```
1. <bean:bean id="chunkHandler"
2.     class="org.springframework.batch.integration.
                           chunk.ChunkProcessorChunkHandler">
3.     <bean:property name="chunkProcessor">
4.         <bean:bean class="org.springframework.batch.core.
                           step.item.SimpleChunkProcessor">
5.             <bean:property name="itemWriter" ref="jdbcItemWriter"/>
6.             <bean:property name="itemProcessor">
7.                 <bean:bean class="org.springframework.batch.item.
                           support.PassThroughItemProcessor"/>
8.             </bean:property>
9.         </bean:bean>
10.    </bean:property>
11.</bean:bean>
```

其中，1~2 行：通过 ChunkProcessorChunkHandler 定义实现类。

3~10 行：定义面向 Chunk 的处理类 SimpleChunkProcessor。

5 行：面向 Chunk 的处理类 SimpleChunkProcessor 定义使用的 itemWriter。

7 行：面向 Chunk 的处理类 SimpleChunkProcessor 定义使用的 itemProcessor。

11.4.2.4 配置远程 Step

上面的配置完成了队列、服务器、本地 Step 的配置后，接下来只需要完成远程 Step 的配置即可；远程 Step 本质上是一个队列的监听器与收到消息后的处理逻辑。配置远程 Step 参见代码清单 11-15。

完整配置参见文件：/ch11/job/job-chunk-remote.xml。

代码清单 11-15 配置远程 Step

```
1. <jms:listener-container connection-factory="connectionFactory"
2.     transaction-manager="transactionManager"
3.     acknowledge="transacted" concurrency="1">
4.     <jms:listener destination="requests" response-destination="replies"
5.         ref="chunkHandler" method="handleChunk" />
6. </jms:listener-container>
```

其中，1~6 行：配置请求队列的监听器与处理逻辑，处理逻辑调用 chunkHandler 的 handleChunk 操作；该操作接受 chunk 请求，并负责处理逻辑。

5 行：定义监听的队列为 requests，监听到消息后处理逻辑为 chunkHandler 的 handleChunk

操作，处理后的消息发送到 replies 队列。

11.4.2.5 Job 配置及运行

接下来我们定义远程 Job，参见代码清单 11-16。

完整配置参见文件：/ch11/job/job-chunk-remote.xml。

代码清单 11-16 定义远程 Job

```
1. <job id="remoteChunkJob">
2.     <step id="stepRemoteChunk">
3.         <tasklet>
4.             <chunk reader="jdbcItemPageReader" writer="jdbcItemWriter"
5.                                     commit-interval="10" />
6.         </tasklet>
7.     </step>
8. </job>
```

本节的 Job 定义和普通的 Job 没有任何区别；通过数据库读、然后再通过数据库写入。jdbcItemPageReader 是从表 t_cretid 读取数据，jdbcItemWriter 将数据写入 t_destcredit 中。为了节省篇幅，jdbcItemPageReader 与 jdbcItemWriter 的具体配置请参见文件/ch11/job/job-chunk-remote.xml。

使用代码清单 11-17 执行定义的 remoteChunkJob。

完整代码参见：test.com.juxtapose.example.ch11.JobLaunchChunkRemote。

代码清单 11-17 执行 remoteChunkJob

```
1. JobLaunchBase.executeJob("ch11/job/job-chunk-remote.xml",
2.     "remoteChunkJob",
3.     new JobParametersBuilder().addDate("date", new Date()));
```

测试用例执行完毕后，可以看到库表 t_destcredit 中的数据被远程写入。

至此，完成了远程 Step 的开发示例。读者可以基于 Spring Batch 提供的远程框架自己实现远程 Step 工作，从而提升作业的处理效率。

11.5 分区 Step

通过将任务进行分区，不同的 Step 处理不同的任务数据达到提高 Job 效率的功能。分区模式需要对数据的结构有一定的了解，如主键的范围、待处理的文件的名字等。这种模式的优点在于分区中每一个元素的处理器都能够像一个普通 Spring Batch 任务的单步一样运行，也不必去实现任何特殊的或是新的模式，来让他们能够更容易配置与测试。分区理论上比远程更有扩展性，因为分区并不存在从一个地方读取所有输入数据并进行序列化的瓶颈。

分区作业典型的可以分成两个处理阶段，数据分区、分区处理；分区作业逻辑结构参见图 11-8。

数据分区：根据特殊的规则（例如：根据文件名称，数据的唯一性标识，或者哈希算法）将数据进行合理地切片，为不同的切片生成数据执行上下文 Execution Context、作业步执行器 Step Execution。可以通过接口 Partitioner 生成自定义的分区逻辑，Spring Batch 批处理框架默认对多文件实现 org.springframework.batch.core.partition.support.MultiResourcePartitioner；读者可以自行扩展接口 Partitioner 来实现自定义的分区逻辑。

分区处理：通过数据分区后，不同的数据已经被分配到不同的作业步执行器中，接下来需要交给分区处理器进行作业，分区处理器可以在本地执行也可以在远程执行被划分的作业。接口 PartitionHandler 定义了分区处理的逻辑，Spring Batch 批处理框架默认实现了本地多线程的分区处理 org.springframework.batch.core.partition.support.TaskExecutorPartitionHandler；读者可以自行扩展接口 PartitionHandler 来实现自定义的分区处理逻辑。

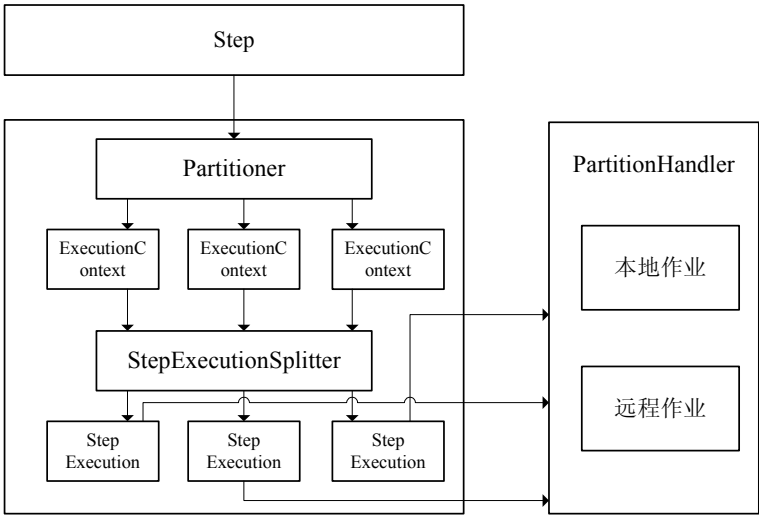


图 11-8 分区作业逻辑结构

11.5.1 关键接口

在 Spring Batch 中有如下接口支持分区：PartitionHandler、StepExecutionSplitter、Partitioner。核心接口关键类图参见图 11-9。

PartitionHandler 知道执行结构-它需要将请求发送到远程步骤并使用任何可以使用的远程技术收集计算结果，PartitionHandler 是一个 SPI，Spring Batch 通过 TaskExecutor 为本地执行提供了一个默认实现，在需要进行有大量 I/O 操作的并发处理时，这个功能是很有用的；Partitioner 接口定义了根据数据结构将作业进行分区，生成执行上下文 Execution Context；StepExecutionSplitter 根据给定 Partitioner 产生的执行上下文生成作业步执行器，然后交给 PartitionHandler 来进行处理。

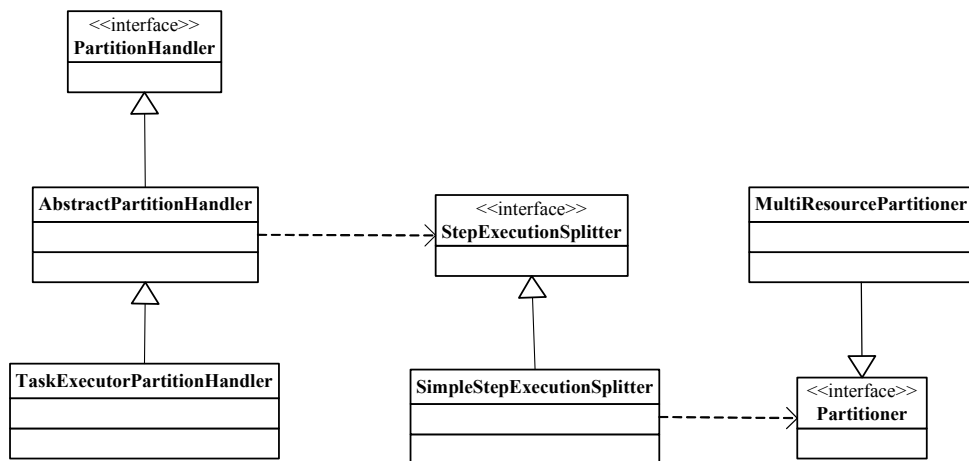


图 11-9 分区关键类图

11.5.1.1 Partitioner

Partitioner 接口定义了如何根据给定的分区规则进行创建作业步执行分区的上下文。每个分区的上下文需要根据对应的分区规则来计算当前分区的处理情况。Partitioner 接口定义参见代码清单 11-18。

代码清单 11-18 Partitioner 接口定义

```

1. public interface Partitioner {
2.     Map<String, ExecutionContext> partition(int gridSize);
3. }
  
```

其中，2 行：操作 partition 根据给定的 gridSize 大小进行执行上下文的划分。

11.5.1.2 StepExecutionSplitter

StepExecutionSplitter 接口定义了如何根据给定的分区规则进行创建作业步执行分区的执行器。StepExecutionSplitter 接口定义参见代码清单 11-19。

代码清单 11-19 StepExecutionSplitter 接口定义

```

1. public interface StepExecutionSplitter {
2.     String getStepName();
3.     Set<StepExecution> split(StepExecution stepExecution, int gridSize)
                           throws JobExecutionException;
4. }
  
```

其中，2 行：操作 getStepName() 获取当前定义的分区作业步的名称。

3 行：操作 split() 根据给定的分区规则为每个分区生成对应的分区执行器。

11.5.1.3 PartitionHandler

PartitionHandler 接口定义了分区处理的逻辑;根据给定的 StepExecutionSplitter 进行分区,并执行,最后将执行的结果进行收集,最终反馈到前端。PartitionHandler 接口定义参见代码清单 11-20。

代码清单 11-20 PartitionHandler 接口定义

```
1. public interface PartitionHandler {
2.     Collection<StepExecution> handle(StepExecutionSplitter stepSplitter,
3.                                     StepExecution stepExecution) throws Exception;
4. }
```

其中,2 行:操作 handle()根据给定的 StepExecutionSplitter 进行分区并执行,最后将执行的结果进行收集,最终反馈到前端。

11.5.2 基本配置

一个典型的分区 Job 配置声明参见代码清单 11-21。

代码清单 11-21 配置典型的分区 Job 示例

```
1. <job id="partitionJob">
2.     <step id="partitionStep">
3.         <partition step="partitionReadWriteStep" partitioner="partitioner">
4.             <handler grid-size="2" task-executor="taskExecutor"/>
5.         </partition>
6.     </step>
7. </job>
8.
9. <step id="partitionReadWriteStep">
10.    <tasklet>
11.        <chunk reader="flatFileItemReader" writer="jdbcItemWriter"
12.            processor="creditBillProcessor" commit-interval="2" />
13.    </tasklet>
14. </step>
15.
16. <bean:bean id="partitioner"
17.    class="org.springframework.batch.core.partition.support.
18.        MultiResourcePartitioner">
19.    <bean:property name="keyName" value="fileName"/>
20.    <bean:property name="resources" value="classpath:/ch11/data/*.csv"/>
21. </bean:bean>
```

在配置分区 Step 之前,我们一起看一下分区 Step 的主要属性定义和元素定义。

图 11-10 展示了分区 Step 属性的 Schema 的定义。

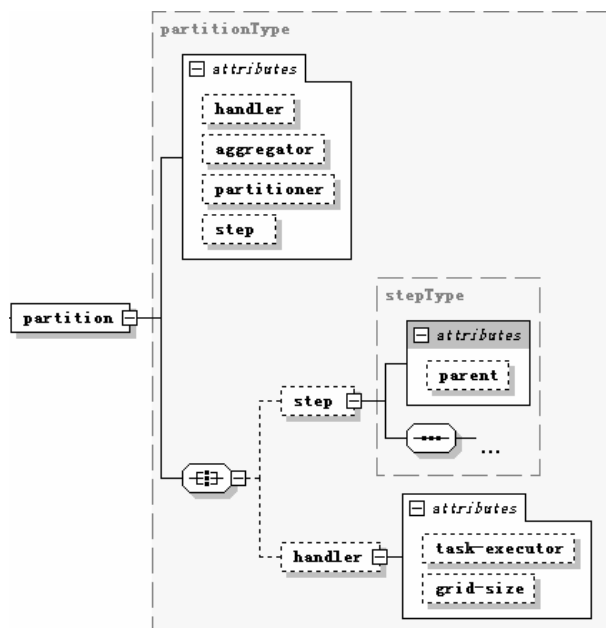


图 11-10 属性 partition 的 Schema 定义

分区 Step 属性说明参见表 11-3。

表 11-3 分区 Step 属性说明

属 性	说 明	默 认 值
step	属性 step 用于指定分区 Step 的名字	
partitioner	属性 partitioner 用于指定当前使用的分区逻辑，需要实现接口 Partitioner	
aggregator	属性 aggregator 用于指定需要使用的聚合器，该聚合器的作用是将各个分区执行器执行的结果汇总到主执行器中，用于统计最终的计算结果，该聚合器需要实现接口 StepExecutionAggregator。	默认使用实现类： DefaultStepExecutionAggregator
handler（属性）	属性 handler 用于指定分区执行器，需要实现接口 PartitionHandler。	
handler（子元素）	用于定义默认的实现： TaskExecutorPartitionHandler	
task-executor	声明使用的线程池	
grid-size	声明分区的 HashMap 的初始值大小	6

接下来的章节，我们给出对文件进行分区和对数据库进行分区的示例，读者可以根据这两个例子扩展出其他可能的分区业务处理。

11.5.3 文件分区

Spring Batch 框架提供了对文件分区的支持，实现类 `org.springframework.batch.core.partition.support.MultiResourcePartitioner` 提供了对文件分区的默认支持，根据文件名将不同的文件处理进行分区，提升处理的速度和效率，适合有大量小文件需要处理的场景。图 11-11 给出了一个多文件的示例，信用卡每月的账单，本章节按照此例子给出如何配置多文件的分区实现。

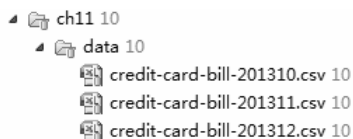


图 11-11 多文件的示例

由于文件数目较多，我们针对文件进行分区，然后将文件的内容写入 DB 中。本节示例的处理逻辑关系参见图 11-12。

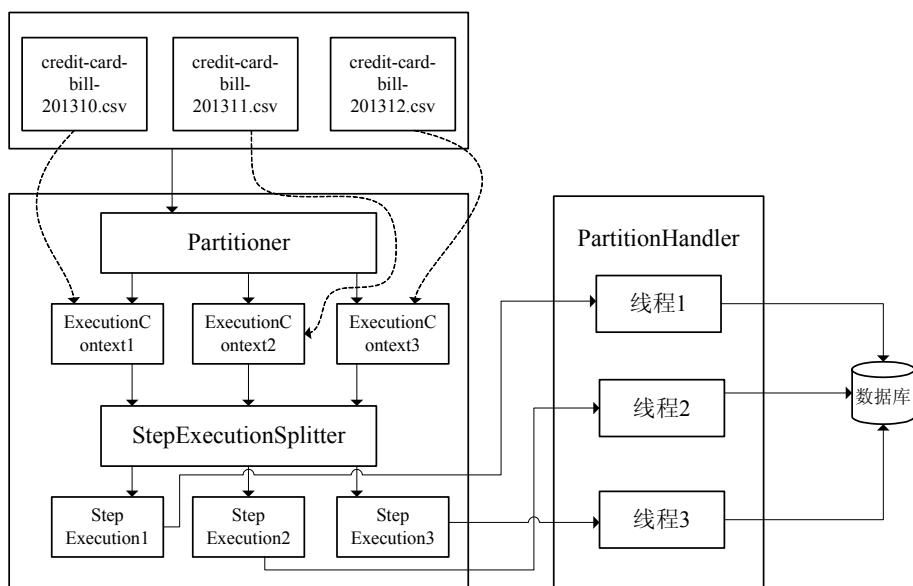


图 11-12 多文件处理逻辑关系图

接下来我们展示如何配置基于文件的分区处理。

配置分区

首先我们定义分区 Step，具体参见代码清单 11-22 配置。

完整配置文件参见：`/ch11/job/job-partition-file.xml`。

代码清单 11-22 配置分区 Step

```
1. <job id="partitionJob">
2.   <step id="partitionStep">
3.     <partition step="partitionReadWriteStep"
4.       partitioner="filePartitioner">
5.       <handler grid-size="2" task-executor="taskExecutor"/>
6.     </partition>
7.   </step>
8. </job>
9. <step id="partitionReadWriteStep">
10.  <tasklet>
11.    <chunk reader="flatFileItemReader" writer="jdbcItemWriter"
12.      processor="creditBillProcessor" commit-interval="2" />
13.    <listeners>
14.      <listener ref="partitionItemReadListener"></listener>
15.    </listeners>
16.  </tasklet>
17. </step>
```

其中，3~5 行：定义了分区作业，属性 `step` 声明了引用的作业步的名称，属性 `partitioner` 引用指定的分区规则，子元素 `handler` 用于指定使用的本地处理的线程池。

9~17 行：具体的作业操作步定义，本例中使用基于文件的读，然后通过 `jdbc` 的方式写入数据库中。

定义分区文件

接下来，我们定义文件分区，将上面的不同文件分配到不同的作业步中，使用 `MultiResourcePartitioner` 进行分区，意味着每个文件会被分配到一个不同的分区中。如果读者有其他的分区规则，可以通过实现接口 `Partitioner` 来进行自定义的扩展。我们展示使用 `MultiResourcePartitioner` 进行默认的分​​区配置，具体参见代码清单 11-23。

代码清单 11-23 配置分区文件

```
1. <bean:bean id="filePartitioner"
2.   class="org.springframework.batch.core.partition.support.
3.     MultiResourcePartitioner">
4.   <bean:property name="keyName" value="fileName"/>
5.   <bean:property name="resources" value="classpath:/ch11/data/*.csv"/>
6. </bean:bean>
```

其中，3 行：属性 `keyName` 用于指定作业步上下文中属性的名字，作用是在不同的作业上下文中可以获取设置的对应属性值，对于 `MultiResourcePartitioner`，对应的值是文件的全路径的名字，可以在对应的读、写等阶段通过 `# {stepExecutionContext[fileName]}` 的方式获取。

4 行：定义需要分区的文件集，使用 `MultiResourcePartitioner` 时候，要求根据统一的后缀名。

定义文件读

配置好分区实现后，我们需要在每个分区的作业步中读入不同的文件，进而提高文件处理的效率。具体参见代码清单 11-24。

代码清单 11-24 配置文件读

```
1. <bean:bean id="flatFileItemReader" scope="step"
2.     class="org.springframework.batch.item.file.FlatFileItemReader">
3.     <bean:property name="resource"
4.         value="#{stepExecutionContext[fileName]}" />
5.     <bean:property name="lineMapper" ref="lineMapper" />
6. </bean:bean>
```

其中，4 行：使用 **filePartitioner** 中定义的属性 **keyName** 获取对应的文件路径作为每个分区作业的文件输入。

定义线程池

分区执行处理器使用本地的线程池来处理不同的分区任务，代码清单 11-25 展示了分区执行器使用的线程池的定义。

代码清单 11-25 配置线程池

```
1. <bean:bean id="taskExecutor"
2.     class="org.springframework.scheduling.concurrent.
3.         ThreadPoolTaskExecutor">
4.     <bean:property name="corePoolSize" value="5"/>
5.     <bean:property name="maxPoolSize" value="15"/>
6. </bean:bean>
```

其中，3 行：属性 **corePoolSize** 定义线程池的始终处于激活状态线程的数量。

4 行：属性 **maxPoolSize** 定义线程池的最大值。

验证分区任务使用不同线程处理

为了更好地给读者验证不同的任务使用了不同的线程处理，本节使用作业步执行拦截器进行验证。在处理分区的作业步之前通过拦截器打印当前的线程名称。

PartitionStepExecutionListener 实现接口 **StepExecutionListener**，在对应的 **beforeStep** 操作中打印当前线程名称。具体的实现代码参见代码清单 11-26。

完成代码参见 `com.juxtapose.example.ch11.partition.PartitionStepExecutionListener`。

代码清单 11-26 PartitionStepExecutionListener 类定义

```
1. public class PartitionStepExecutionListener implements
2.     StepExecutionListener {
3.     @Override
4.     public void beforeStep(StepExecution stepExecution) {
5.         System.out.println("ThreadName=" + Thread.currentThread().
```

```

        getName() + "; "
5.         + "StepName=" + stepExecution.getStepName() + "; "
6.         + "FileName="
7.         + stepExecution.getExecutionContext().getString("fileName"));
8.     }
9.
10.    @Override
11.    public ExitStatus afterStep(StepExecution stepExecution) {
12.        return null;
13.    }
14. }

```

其中，4~7 行：打印出当前作业步的线程名，作业步的名称，对应处理的文件名。

使用代码清单 11-27 执行定义的 `partitionJob`。

完整代码参见：`test.com.juxtapose.example.ch11.JobLaunchPartitionFile`。

代码清单 11-27 执行 `partitionJob`

```

1. public static void main(String[] args) {
2.     JobLaunchBase.executeJob("ch11/job/job-partition-file.xml",
3.         "partitionJob",
4.         new JobParametersBuilder().addDate("date", new Date()));
5. }

```

代码清单 11-28 给出了执行的结果，可以看出不同的文件由不同的线程来处理，并且被分配到不同的分区的作业步中执行。

代码清单 11-28 执行 `partitionJob` 控制台输出

```

1. ThreadName=taskExecutor-2; StepName=partitionReadWriteStep:partition0;
   FileName=file:/...../credit-card-bill-201310.csv
2. ThreadName=taskExecutor-3; StepName=partitionReadWriteStep:partition2;
   FileName=file:/...../credit-card-bill-201312.csv
3. ThreadName=taskExecutor-1; StepName=partitionReadWriteStep:partition1;
   FileName=file:/...../credit-card-bill-201311.csv

```

从控台的输出，我们可以抽取下面的关键信息：

taskExecutor-2--> partitionReadWriteStep:partition0--> credit-card-bill-201310.csv

taskExecutor-3--> partitionReadWriteStep:partition2--> credit-card-bill-201312.csv

taskExecutor-1--> partitionReadWriteStep:partition1--> credit-card-bill-201311.csv

每个不同的线程对文件进行了分区处理。

至此，我们完成了文件分区的处理。有兴趣的读者可以自行阅读文件分区默认实现类 `org.springframework.batch.core.partition.support.MultiResourcePartitioner`，实现非常简单；根据此类的实现，我们可以轻松地模拟出其他的分区实现方式。下一节我们将带领读者通过 `Partitioner` 来实现数据库的分区处理。

11.5.4 数据库分区

本节通过实现接口 `Partitioner` 进行数据分区，我们将对读取数据库表进行分区处理，交给不同的作业步进行处理。本节对数据表 `t_credit` 读取，然后写入表 `t_destcredit` 中。数据表 `t_credit` 中数据非常庞大，正常的单线程的读/写会消耗大量时间处理操作，我们接下来使用分区的技术将任务进行切分，交由不同的分区 `Step` 进行处理。

未分区的批处理作业调度情况参见图 11-13。

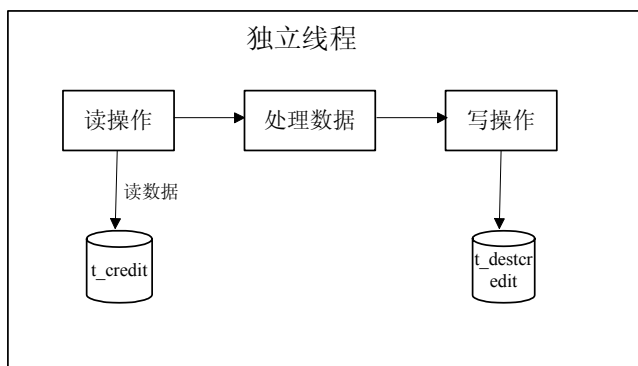


图 11-13 未分区的批处理作业调度

分区后的作业调度情况参见图 11-14，`DBPartitioner` 将表 `t_credit` 分割为 3 个作业步处理，每个作业步由独立的线程进行处理，提高了处理的效率。

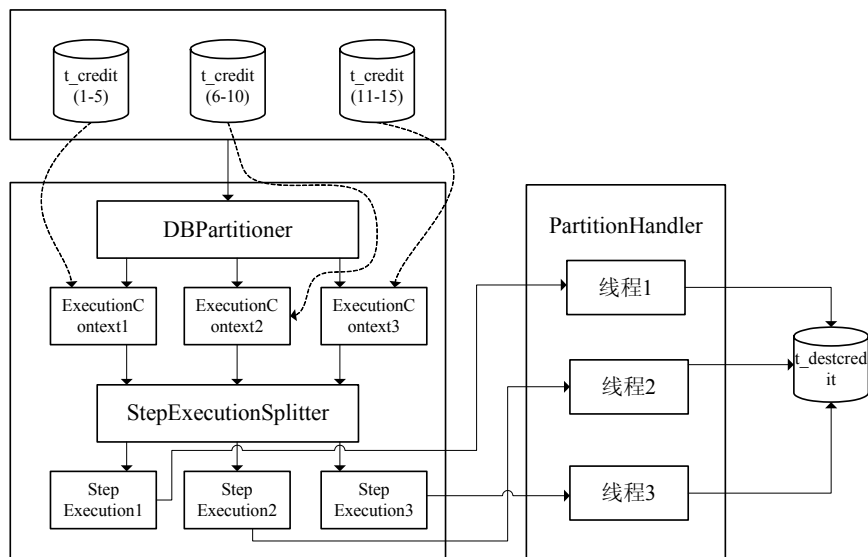


图 11-14 分区后批处理作业调度

实现 DBPartitioner

DBpartition 实现接口 Partitioner (参见代码清单 11-29), 其主要操作 partition(int gridSize) 需要完成的任务非常简单, 只需要根据给定的分区 gridSize 的大小, 对数据进行切片, 并为每个分区的切片生成对应的执行上下文 ExecutionContext。在本例中, 分区上下文 ExecutionContext 中存放数据片段的开始值 _minRecord 和数据片段的结束值 _maxRecord。在后续每个分区片段执行的过程中可以从对应的分区上下文中获取数据分段的开始和结束值 _minRecord 与 _maxRecord。

可以通过 #stepExecutionContext[_minRecord] 获取数据片段的开始值, 通过 #stepExecutionContext[_maxRecord] 获取数据片段的最大值。

完整代码参见: com.juxtapose.example.ch11.partition.db.DBpartition。

代码清单 11-29 DBpartition 类定义

```
1. public class DBpartition implements Partitioner {
2.     private static final String _MINRECORD = "_minRecord";
3.     private static final String _MAXRECORD = "_maxRecord";
4.     private static final String MIN_SELECT_PATTERN = "select min({0}) from {1}";
5.     private static final String MAX_SELECT_PATTERN = "select max({0}) from {1}";
6.     private JdbcTemplate jdbcTemplate ;
7.     private DataSource dataSource;
8.     private String table ;
9.     private String column;
10.
11.     public Map<String, ExecutionContext> partition(int gridSize) {
12.         validateAndInit();
13.         Map<String, ExecutionContext> resultMap =
14.             new HashMap<String, ExecutionContext>();
15.         int min = jdbcTemplate.queryForInt(MessageFormat.
16.             format(MIN_SELECT_PATTERN, new Object[]{column,table}));
17.         int max = jdbcTemplate.queryForInt(MessageFormat.
18.             format(MAX_SELECT_PATTERN, new Object[]{column,table}));
19.         int targetSize = (max-min)/gridSize +1;
20.         int number=0;
21.         int start =min;
22.         int end = start+targetSize-1;
23.         while(start <= max){
24.             ExecutionContext context = new ExecutionContext();
25.             if(end>=max){ end=max;}
26.             context.putInt(_MINRECORD, start);
27.             context.putInt(_MAXRECORD, end);
28.             start+=targetSize;
29.             end+=targetSize;
30.             resultMap.put("partition"+(number++), context);
31.         }
32.     }
33. }
```



```

28.     }
29.     return resultMap;
30. }
31. }

```

其中，14 行：根据主键计算最小值。

15 行：根据主键计算最大值。

20~28 行：根据给定的 gridSize 大小将数据进行分区，根据对应的主键进行平均划分。

23~24 行：将 _minRecord 与 _maxRecord 放入作业步上下文中。

配置分区

首先我们定义分区 Step，具体参见代码清单 11-30 配置。

完整配置文件参见：/ch11/job/job-partition-db.xml。

代码清单 11-30 配置分区

```

1. <job id="partitionJob" restartable="true">
2.   <step id="partitionStep">
3.     <partition step="partitionReadWriteDB" partitioner="partitionerDB">
4.       <handler grid-size="3" task-executor="taskExecutor"/>
5.     </partition>
6.   </step>
7. </job>
8.
9. <step id="partitionReadWriteDB">
10.  <tasklet>
11.    <chunk reader="jdbcItemPageReader" writer="jdbcItemWriter"
12.      processor="creditBillProcessor" commit-interval="2"/>
13.    <listeners>
14.      <listener ref="partitionItemReadListener"></listener>
15.    </listeners>
16.  </tasklet>
17. </step>

```

其中，3~5 行：定义了分区作业，属性 step 声明了引用的作业步的名称，属性 partitioner 引用指定的分区规则，子元素 handler 用于指定使用的本地处理的线程池。

9~17 行：具体的作业操作步定义，本例中使用基于数据库的读，然后通过 jdbc 的方式写入数据库中。

定义分区数据文件

接下来，我们定义文件分区，将上面的数据片段分配到不同的作业步中，使用 DBPartitioner 进行分区，意味着每个数据片段会被分配到一个不同的分区中。具体配置参见代码清单 11-31。

代码清单 11-31 配置分区数据文件

```
1. <!-- db 数据切分 -->
2. <bean:bean id="partitionerDB"
3.     class="com.juxtaPOSE.example.ch11.partition.db.DBpartition">
4.     <bean:property name="table" value="t_credit"/>
5.     <bean:property name="column" value="ID"/>
6.     <bean:property name="dataSource" ref="dataSource"/>
7. </bean:bean>
```

其中，4 行：属性 **table** 指定分区的表，本例使用 **t_credit**。

5 行：属性 **column** 指定分区使用的主键字段，本例使用 **ID** 字段。

6 行：属性 **dataSource** 指定使用的数据源。

定义数据库读

配置好分区实现后，我们需要在每个分区的作业步中读入不同的数据片段，进而提高数据处理的效率。具体配置数据库参见代码清单 11-32。

代码清单 11-32 配置数据库度

```
1. <!-- 从 db 分页读数据 -->
2. <bean:bean id="jdbcItemPageReader" scope="step"
3.     class="org.springframework.batch.item.database.JdbcPaging
4.         ItemReader">
5.     <bean:property name="dataSource" ref="dataSource"/>
6.     <bean:property name="queryProvider" ref="refQueryProvider" />
7.     <bean:property name="pageSize" value="2"/>
8.     <bean:property name="rowMapper" ref="custCreditRowMapper"/>
9. </bean:bean>
10.
11. <bean:bean id="refQueryProvider" scope="step"
12.     class="org.springframework.batch.item.database.support
13.         .SqlPagingQueryProviderFactoryBean">
14.     <bean:property name="dataSource" ref="dataSource"/>
15.     <bean:property name="selectClause"
16.         value="select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS"/>
17.     <bean:property name="fromClause" value="from t_credit"/>
18.     <bean:property name="whereClause"
19.         value="where ID between #{stepExecutionContext[_minRecord]}
20.             and #{stepExecutionContext[_maxRecord]}/>
21.     <bean:property name="sortKey" value="ID"/>
22. </bean:bean>
```

其中，15 行：为读操作指定分区的开始标志 **_minRecord** 和结束标志 **_maxRecord**。

定义线程池

分区执行处理器使用本地的线程池来处理不同的分区任务，代码清单 11-33 展示了分区执行器使用的线程池的定义。

代码清单 11-33 配置线程池

```
1. <bean:bean id="taskExecutor"  
2.     class="org.springframework.scheduling.concurrent.  
    ThreadPoolTaskExecutor">  
3.     <bean:property name="corePoolSize" value="5"/>  
4.     <bean:property name="maxPoolSize" value="15"/>  
5. </bean:bean>
```

其中，3 行：属性 `corePoolSize` 定义线程池的始终处于激活状态线程的数量。

4 行：属性 `maxPoolSize` 定义线程池的最大值。

验证分区任务使用不同线程处理

为了更好地给读者验证不同的任务使用了不同的线程处理，本节使用作业步执行拦截器进行验证。在处理分区的作业步之前通过拦截器打印当前的线程名称。

`PartitionStepExecutionListener` 实现接口 `StepExecutionListener`，在对应的 `beforeStep` 操作中打印当前线程名称。具体的实现代码参见代码清单 11-34。

完成代码参见 `com.juxtapose.example.ch11.partition.db.PartitionStepExecutionListener`。

代码清单 11-34 `PartitionStepExecutionListener` 类定义

```
1. public class PartitionStepExecutionListener implements  
    StepExecutionListener {  
2.     @Override  
3.     public void beforeStep(StepExecution stepExecution) {  
4.         System.out.println("ThreadName="+Thread.currentThread().  
            getName() + " ;"  
5.             + "StepName=" + stepExecution.getStepName() + " ;");  
6.     }  
7.  
8.     @Override  
9.     public ExitStatus afterStep(StepExecution stepExecution) {  
10.         return null;  
11.     }  
12. }
```

其中，4~7 行：打印出当前作业步的线程名，作业步的名称，对应处理的文件名。

使用代码清单 11-35 执行定义的 `partitionJob`。

完整代码参见：`test.com.juxtapose.example.ch11.JobLaunchPartitionDB`。

代码清单 11-35 执行 partitionJob

```
1. public static void main(String[] args) {
2.     JobLaunchBase.executeJob("ch11/job/job-partition-db.xml",
        "partitionJob",
3.         new JobParametersBuilder().addDate("date", new Date()));
4. }
```

代码清单 11-36 给出了执行的结果，可以看出表中不同的数据由不同的线程来处理，并且被分配到不同的分区的作业步中执行。

代码清单 11-36 执行 partitionJob 控制台输出

```
1. ThreadName=taskExecutor-3; StepName=partitionReadWriteDB:partition1;
2. ThreadName=taskExecutor-2; StepName=partitionReadWriteDB:partition2;
3. ThreadName=taskExecutor-1; StepName=partitionReadWriteDB:partition0;
```

从控制台输出，我们可以抽取下面的关键信息：

taskExecutor-1--> partitionReadWriteDB:partition0

taskExecutor-2--> partitionReadWriteDB:partition2

taskExecutor-3--> partitionReadWriteDB:partition1

每个不同的线程对数据文件进行了分区处理。

11.5.5 远程分区 Step

前面章节我们介绍了远程 Step 和分区 Step，本节我们将提供一个远程分区 Step 的例子供读者参考。

远程分区 Job 对本地的多文件进行分区读取，然后通过远程的方式发送到远端执行，通过 JDBC 的方式写入数据库中。远程分区的处理示意图参见图 11-15。

通过分区 Partitioner 将多文件进行切分，通过 StepExecutionSplitter 将资源进行分割成不同的作业步执行上下文；MessageChannelPartitionHandler 将作业步执行上下文通过消息的方式发送到请求队列，远程机器的拦截器监听队列消息的到达，收取消息后交给 ChunkProcessor ChunkHandler 处理，最终通过 JDBC 的方式写入数据库的 t_destcredit 表中；处理完成后发送响应消息到响应队列，MessageChannelPartitionHandler 负责收取响应队列的消息，将远端的执行情况进行汇总。

11.5.5.1 配置远程分区 Job

代码清单 11-37 定义了远程分区 Job。

完整配置文件参见：ch11/job/job-partition-remote.xml。

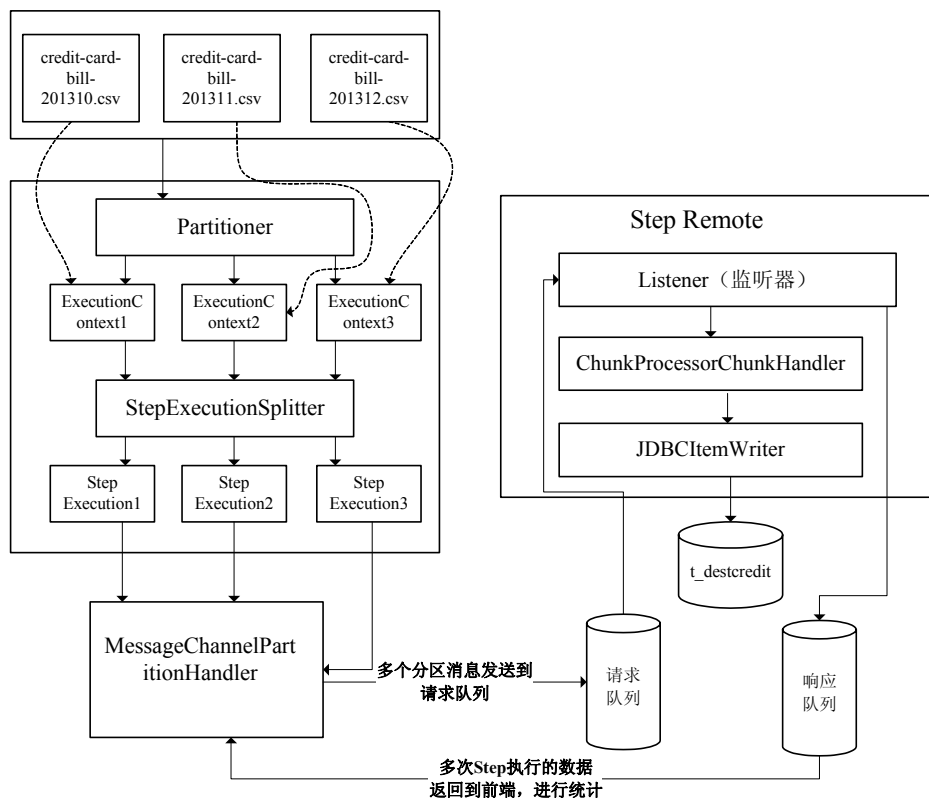


图 11-15 远程分区处理示意图

代码清单 11-37 配置远程分区 partitionRemoteJob

```

1. <job id="partitionRemoteJob">
2.   <step id="partitionRemoteStep">
3.     <partition partitioner="partitioner" handler="partitionHandler" />
4.   </step>
5. </job>

```

其中，3 行：属性 partitioner 定义分区定义，属性 partitionHandler 定义分区后的处理逻辑。

11.5.5.2 配置本地 Step

本地 Step 配置参见代码清单 11-38。

完整配置文件参见：ch11/job/job-partition-remote.xml。

代码清单 11-38 配置本地 Step

```

1. <bean:bean id="partitioner"
2.   class="org.springframework.batch.core.partition.
      support.MultiResourcePartitioner">

```

```

3.     <bean:property name="keyName" value="fileName"/>
4.     <bean:property name="resources" value="classpath:/ch11/data/*.csv"/>
5. </bean:bean>
6.
7. <bean:bean id="partitionHandler"
8.     class="org.springframework.batch.integration.
           partition.MessageChannelPartitionHandler">
9.     <bean:property name="messagingOperations">
10.        <bean:bean class="org.springframework.integration.
                  core.MessagingTemplate">
11.            <bean:property name="defaultChannel" ref="requests" />
12.            <bean:property name="receiveTimeout" value="30000" />
13.        </bean:bean>
14.    </bean:property>
15.    <bean:property name="replyChannel" ref="replies"/>
16.    <bean:property name="stepName" value="remoteStep" />
17.    <bean:property name="gridSize" value="2" />
18. </bean:bean>
19.
20. <step id="remoteStep">
21.     <tasklet>
22.         <chunk reader="flatFileItemReader" writer="jdbcItemWriter"
                  commit-interval="10"/>
23.         <listeners>
24.             <listener ref="partitionItemReadListener"></listener>
25.         </listeners>
26.     </tasklet>
27. </step>

```

其中，1~5 行：定义了分区策略，基于文件的分区，/ch11/data/*.csv 下面的所有文件单独划分为一个分区。

7~18 行：定义分区后的处理逻辑，messagingOperations 定义了消息发送接口，默认的发送消息到 requests 队列。

15 行：属性 replyChannel 定义消息返回的队列为 replies。

16 行：定义远程执行的作业步，定义真正的对文件读/写的逻辑。

20~27 行：定义远端执行的作业步 remoteStep，基于文件的读，基于 JDBC 的数据库写。

23~25 行：定义作业步 remoteStep 的拦截器，主要用于打印执行的线程名与对应处理的分区文件；完整的拦截器代码参见：com.juxtapose.example.ch11.partition.PartitionStepExecution Listener。

11.5.5.3 配置远程 Step

远程 Step 本质上是一个队列的监听器与收到消息后的处理逻辑。具体配置参见代码

清单 11-39。

完整配置参见文件：/ch11/job/job-partition-remote.xml。

代码清单 11-39 配置远程 Step

```
1. <jms:listener-container connection-factory="connectionFactory"
2.     transaction-manager="transactionManager"
3.     acknowledge="transacted" concurrency="10">
4.     <jms:listener destination="requests" response-destination="replies"
5.         ref="stepExecutionRequestHandler" method="handle" />
6. </jms:listener-container>
7.
8. <bean:bean id="stepExecutionRequestHandler"
9.     class="org.springframework.batch.integration.
10.         partition.StepExecutionRequestHandler">
11.     <bean:property name="jobExplorer" ref="jobExplorer"/>
12.     <bean:property name="stepLocator" ref="stepLocator"/>
13. </bean:bean>
14. <bean:bean id="stepLocator"
15.     class="org.springframework.batch.integration.
16.         partition.BeanFactoryStepLocator" />
```

其中，1~6 行：配置请求队列的监听器与处理逻辑，处理逻辑调用 `stepExecutionRequestHandler` 的 `handle` 操作；该操作接受 `chunk` 请求，并负责处理逻辑。

4 行：定义监听的队列为 `requests`，监听到消息后处理逻辑为 `stepExecutionRequestHandler` 的 `handle` 操作，处理后的消息发送到 `replies` 队列。

8~12 行：定义接受消息后的处理逻辑，通过给定的属性 `jobExplorer` 和 `stepLocator` 在 `jobRepository` 查找到指定的作业步；通过 `jobExplorer` 接口，可以进行作业状态查询类，返回返回作业状态的复杂对象，如 `Job Execution`、`Job Instance`、`StepExecution` 等。

13~14 行：`stepLocator` 定义了作业步查找类，通过 `stepName` 可以从上下文中找到对应的 `Step` 对象。

11.5.5.4 运行远程分区 Job

在能够顺利执行远程分区 `Job` 之前，需要定义消息队列及配置 `AMQ` 服务器，读者可以参考 11.4.2 节中的配置消息队列、配置 `AMQ` 服务器章节。完整的配置文件参见：/ch11/job/job-partition-remote.xml。

使用代码清单 11-40 执行定义的 `partitionRemoteJob`。

完整代码参见：`test.com.juxtapose.example.ch11.JobLaunchPartitionRemote`。

代码清单 11-40 执行 `partitionRemoteJob`

```
1. JobLaunchBase.executeJob("ch11/job/job-partition-remote.xml",
2.     "partitionRemoteJob",
3.     new JobParametersBuilder().addDate("date", new Date()));
```

验证点 1：数据被正确地写入数据库。

测试用例执行完毕后，可以看到库表 `t_destcredit` 中的数据被远程写入。

验证点 2：文件分区及被不同的线程处理。

代码清单 11-41 给出了执行的结果，可以看出不同的文件由不同的线程来处理，并且被分配到不同的分区的作业步中执行。

代码清单 11-41 执行 `partitionRemoteJob` 控制台输出

```
1. ThreadName=org.springframework.jms.listener.DefaultMessage
   ListenerContainer#0-2; StepName=partitionRemoteStep:partition0;
   FileName= file:/...../data/credit-card-bill-201310.csv
2. ThreadName=org.springframework.jms.listener.DefaultMessageListener
   Container#0-1; StepName=partitionRemoteStep:partition2; FileName=file:
   /...../ch11/data/credit-card-bill-201312.csv
3. ThreadName=org.springframework.jms.listener.DefaultMessageListener
   Container#0-3; StepName=partitionRemoteStep:partition1; FileName=file:
   /...../ch11/data/credit-card-bill-201311.csv
```

验证点 3：查看数据库表 `batch_step_execution` 中的作业步信息，参见图 11-16、图 11-17。

	STEP_EXECUTION_ID	VERSION	STEP_NAME	STATUS
1	811	2	partitionRemoteStep	COMPLETED
2	812	3	partitionRemoteStep:partition2	COMPLETED
3	813	3	partitionRemoteStep:partition1	COMPLETED
4	814	3	partitionRemoteStep:partition0	COMPLETED

图 11-16 数据库表 `batch_step_execution` 中的作业步信息（一）

COMMIT_COUNT	READ_COUNT	WRITE_COUNT	START_TIME	END_TIME
3	18	18	2014-03-29 21:00:56	2014-03-29 21:00:59
1	6	6	2014-03-29 21:00:57	2014-03-29 21:00:57
1	6	6	2014-03-29 21:00:57	2014-03-29 21:00:57
1	6	6	2014-03-29 21:00:57	2014-03-29 21:00:57

图 11-17 数据库表 `batch_step_execution` 中的作业步信息（二）

通过上面的数据库信息可以看出，作业步 `partitionRemoteStep` 被划分为了三个分区，分别是 `partitionRemoteStep:partition0`、`partitionRemoteStep:partition1`、`partitionRemoteStep:partition2`；每个分区的作业步各处理 6 条消息，最后作业步 `partitionRemoteStep` 成功读/写了 18 条消息，正好是每个分区作业处理消息条数的总和。

后 记

随着近年来互联网应用、云计算、移动互联网的发展，大数据的处理已成为当前企业迫切需要面对的问题。数据批量操作、数据分析、数据处理、数据抽取、元数据、数据质量、数据仓库、数据集成等一系列数据的概念如雨后春笋般地涌现，各种批处理的工具、数据集成框架、ETL 工具在开源社区也纷纷涌现。Spring Batch 就是在这样背景下出现的一款批处理框架。

Spring Batch 是一款基于 Spring 的企业批处理框架。通过它可以构建出健壮的企业批处理应用。Spring Batch 不仅提供了统一的读/写接口、丰富的任务处理方式、灵活的事务管理、并发处理，同时还支持日志、监控、任务重启与跳过等特性，大大简化了批处理应用开发，将开发人员从复杂的任务配置管理过程中解放出来，使他们可以更多地去关注核心的业务处理过程。

2007 年，Spring Batch 项目进入开发阶段；2008 年 3 月，Spring Batch 发布第一个正式版本 1.0.0；2014 年 10 月推出 Spring Batch 2.2.7 版本。2013 年 4 月发布的 JSR-352 标准定义了 Java 平台上的批处理应用程序，为应用开发人员提供了一个开发健壮批处理系统的模型，该模型的核心便借鉴了 Spring Batch 开发模式。Reader-Processor-Writer 模式，在这个模型中鼓励开发人员遵循面向块的处理标准。能够在 JSR-352 标准中看到 Spring Batch 框架的身影，表明 Spring Batch 框架在批处理的架构设计上得到了充分的认可。

但在企业级应用中面对批量数据处理，提供批处理框架仅能满足批处理作业的快速开发、执行能力。企业需要统一的批处理平台来处理复杂的企业批处理应用，批处理平台需要解决作业的统一调度、批处理作业的集中管理和管控和批处理作业的统一监控。试想一下，目前国内一般的银行每日处理的批处理作业达到万量级的水平，国内互联网每日的作业量在百万量级的水平，如果没有友好的统一作业调度平台进行支撑是一件多么可怕的事情。企业级的批处理平台需要在 Spring Batch 批处理框架的基础上集成调度框架，通过调度框架可以将任务按照企业的需求进行任务的定期执行；丰富目前 Spring Batch Admin（Spring Batch 的管理监控平台，目前能力还比较薄弱）框架，提供对 Job 的统一管理功能，增强 Job 作业的监控、预警等能力；通过与企业的组织机构、权限管理、认证系统进行合理的集成，增强平台对 Job 作业的权限控制、安全管理能力。

Spring Batch 框架针对批处理应用的支持在框架级别已经做到了完美，故本书就是将 Spring Batch 框架带给国内的读者，希望有更多的开源产品或者企业批处理平台能够基于 Spring Batch 进行发展，进一步带动 Spring Batch 的生态圈。

编者于上海
2014 年 12 月